

Arm[®] Architecture Reference Manual

for R-profile AArch32 architecture

Document number	DDI 111353
Document quality	EAC
Document version	A.a
Document confidentiality	Non-Confidential
Date of issue	09 July 2026



Release information

The following releases of this document have been made.

Date	Version	Changes
09/Jul/2026	A.a	Initial release of Non-Confidential Arm Architecture Reference Manual for R-profile AArch32 architecture with ASLv1.0

Non-Confidential Proprietary Notice

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

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

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

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

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

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

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

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

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

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

Copyright © 2016, 2017, 2020, 2025-2026 Arm Limited or its affiliates. All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-20349

8 March 2024

Confidentiality Status

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

Product Status

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

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

Web Address

<https://www.arm.com>

Contents

Arm[®] Architecture Reference Manual, for R-profile AArch32 architecture

Preface

About this Manual	14
Using this Manual.....	16
Conventions	20
Additional reading.....	23
Feedback	24

Part A Arm Architecture Introduction and Overview

Chapter A1 Introduction to the Arm Architecture

A1.1 About the Arm architecture	27
A1.2 Architecture profiles	28
A1.3 Arm architectural concepts.....	29
A1.4 Supported data types	32
A1.5 Floating-point support.....	40
A1.6 The Arm memory model.....	48

Chapter A2 R-profile Architecture Extensions

A2.1 About the Armv8-R AArch32 architecture extensions.....	50
A2.2 Armv8-R AArch32 architecture extensions	51

Part B The AArch32 Application Level Architecture

Chapter B1 The AArch32 Application Level Programmers' Model

B1.1 About the Application level programmers' model	57
B1.2 The Application level programmers' model in AArch32 state.....	58
B1.3 Advanced SIMD and floating-point instructions	69
B1.4 About the AArch32 System register interface	85
B1.5 Exceptions	86

Chapter B2 The AArch32 Application Level Memory Model

B2.1 About the Arm memory model	88
B2.2 Atomicity in the Arm architecture	90
B2.3 Definition of the memory model.....	94

B2.4	Ordering requirements defined by the formal concurrency model	95
B2.5	Additional ordering requirements outside of the scope of the formal concurrency model	126
B2.6	Endian support	133
B2.7	Caches and memory hierarchy	137
B2.8	Alignment support	142
B2.9	Memory types and attributes	144
B2.10	Mismatched memory attributes	154
B2.11	Synchronization and semaphores	157

Part C

The AArch32 Instruction Sets

Chapter C1

About the T32 and A32 Instruction Descriptions

C1.1	Format of instruction descriptions	168
C1.2	Standard assembler syntax fields	173
C1.3	Conditional execution	174
C1.4	Shifts applied to a register	176
C1.5	Memory accesses	178
C1.6	Encoding of lists of general-purpose registers and the PC	179
C1.7	General information about the T32 and A32 instruction descriptions	180
C1.8	Additional pseudocode support for instruction descriptions	194
C1.9	Additional information about Advanced SIMD and floating-point instructions	195

Chapter C2

The AArch32 Instruction Sets Overview

C2.1	Support for instructions in different versions of the Arm architecture	202
C2.2	Unified Assembler Language	203
C2.3	Branch instructions	205
C2.4	Data-processing instructions	206
C2.5	PSTATE and banked register access instructions	214
C2.6	Load/store instructions	215
C2.7	Load/store multiple instructions	217
C2.8	Miscellaneous instructions	218
C2.9	Exception-generating and exception-handling instructions	219
C2.10	System register access instructions	221
C2.11	Advanced SIMD and floating-point load/store instructions	222
C2.12	Advanced SIMD and floating-point register transfer instructions	224
C2.13	Advanced SIMD data-processing instructions	225
C2.14	Floating-point data-processing instructions	233

Chapter C3

T32 Instruction Set Encoding

C3.1	T32 instruction set encoding	235
C3.2	About the T32 Advanced SIMD and floating-point instructions and their encoding	307
Chapter C4	A32 Instruction Set Encoding	
C4.1	A32 instruction set encoding	309
C4.2	About the A32 Advanced SIMD and floating-point instructions and their encoding	372
Chapter C5	T32 and A32 Base Instruction Set Instruction Descriptions	
C5.1	Alphabetical list of T32 and A32 base instruction set instructions	374
C5.2	Encoding and use of banked register transfer instructions.....	1138
Chapter C6	T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions	
C6.1	Alphabetical list of Advanced SIMD and floating-point instructions.....	1143
Part D	The AArch32 System Level Architecture	
Chapter D1	The AArch32 System Level Programmers' Model	
D1.1	About the AArch32 System level programmers' model	1890
D1.2	Exception levels.....	1891
D1.3	Exception terminology	1892
D1.4	Execution state.....	1894
D1.5	Instruction Set state	1895
D1.6	Virtualization	1896
D1.7	AArch32 state PE modes	1898
D1.8	AArch32 general-purpose registers, the PC, and the Special-purpose registers	1901
D1.9	Process state, PSTATE.....	1904
D1.10	Instruction set states	1909
D1.11	Handling exceptions.....	1911
D1.12	Routing of aborts	1925
D1.13	Exception return to an Exception level using AArch32	1926
D1.14	Asynchronous exception behavior for exceptions taken from AArch32 state.....	1930
D1.15	AArch32 state exception descriptions.....	1933
D1.16	Reset into AArch32 state.....	1947
D1.17	Mechanisms for entering a low-power state	1948
D1.18	The AArch32 System register interface	1953
D1.19	Advanced SIMD and floating-point support.....	1956
D1.20	Configurable instruction controls.....	1960
Chapter D2	AArch32 Self-hosted Debug	
D2.1	About self-hosted debug	1965

D2.2	Routing debug exceptions	1969
D2.3	The debug exception enable controls.....	1970
D2.4	The effect of powerdown on debug exceptions	1971
D2.5	Summary of permitted routing and enabling of debug exceptions.....	1972
D2.6	Pseudocode description of debug exceptions	1973
D2.7	Breakpoint Instruction exceptions	1974
D2.8	Breakpoint exceptions	1977
D2.9	Watchpoint exceptions.....	1996
D2.10	Vector Catch exceptions.....	2008
D2.11	Synchronization and debug exceptions	2013
Chapter D3	AArch32 Trace	
D3.1	About trace	2016
D3.2	Prohibited regions in self-hosted trace.....	2017
D3.3	Self-hosted trace timestamps	2018
D3.4	Synchronization in self-hosted trace	2019
Chapter D4	The AArch32 System Level Memory Model	
D4.1	About the memory system architecture.....	2021
D4.2	Address space	2022
D4.3	Mixed-endian support in AArch32	2023
D4.4	AArch32 cache and branch predictor support.....	2024
D4.5	System register support for IMPLEMENTATION DEFINED memory features	2044
D4.6	External aborts	2045
D4.7	Memory barrier instructions.....	2047
D4.8	Pseudocode description of general memory System instructions	2048
Chapter D5	The AArch32 Protected Memory System Architecture	
D5.1	About the AArch32 Protected Memory System Architecture	2052
D5.2	Memory protection units	2054
D5.3	Default memory maps and Background region checks.....	2057
D5.4	Protection region attributes and access permissions	2059
D5.5	PMSAv8-32 memory aborts	2063
D5.6	Exception reporting in a PMSAv8-32 implementation.....	2069
D5.7	PMSAv8-32 implications for caches.....	2080
Chapter D6	The Performance Monitors Extension	
D6.1	About the Performance Monitors	2082
D6.2	Accuracy of the Performance Monitors	2084
D6.3	Behavior on overflow.....	2086

D6.4	Attributability	2088
D6.5	Controlling the PMU counters	2089
D6.6	Multithreaded implementations.....	2092
D6.7	Event filtering	2093
D6.8	Performance Monitors and Debug state.....	2095
D6.9	Counter access	2096
D6.10	Performance Monitors Extension registers	2097
Chapter D7	PMU Event Descriptions	
D7.1	Definitions	2099
D7.2	The PMU event number space and common events	2108
D7.3	Common event numbers	2151
D7.4	Cycle event counting.....	2309
D7.5	Meaningful combinations of common events	2311
D7.6	Required events	2315
D7.7	IMPLEMENTATION DEFINED event numbers.....	2316
Chapter D8	The Generic Timer in AArch32 state	
D8.1	About the Generic Timer in AArch32 state.....	2318
D8.2	The AArch32 view of the Generic Timer.....	2322
Chapter D9	AArch32 System Register Encoding	
D9.1	The AArch32 System register encoding space	2328
D9.2	Organization of registers in the (<code>coproc == 0b1110</code>) encoding space.....	2329
D9.3	Organization of registers in the (<code>coproc == 0b1111</code>) encoding space.....	2332
Chapter D10	AArch32 System Register Descriptions	
D10.1	About the AArch32 System registers	2341
D10.2	General system control registers	2355
D10.3	Debug System registers.....	2776
D10.4	Performance Monitors registers.....	2861
D10.5	Generic Timer registers.....	2920
Part E	External Debug	
Chapter E1	About External Debug	
E1.1	Introduction to external debug.....	2962
E1.2	External debug	2963
E1.3	Required debug authentication.....	2964
Chapter E2	Debug State	

E2.1	About Debug state.....	2966
E2.2	Halting the PE on debug events	2967
E2.3	Entering Debug state	2973
E2.4	Behavior in Debug state	2976
E2.5	Exiting Debug state	2992
Chapter E3	Halting Debug Events	
E3.1	Introduction to Halting debug events	2995
E3.2	Halting Step debug events	2997
E3.3	Halt Instruction debug event	3006
E3.4	Exception Catch debug event	3007
E3.5	External Debug Request debug event	3010
E3.6	OS Unlock Catch debug event	3012
E3.7	Reset Catch debug events	3013
E3.8	Software Access debug event	3014
E3.9	Synchronization and Halting debug events	3015
Chapter E4	The Debug Communication Channel and Instruction Transfer Register	
E4.1	Introduction	3017
E4.2	DCC and ITR registers	3018
E4.3	DCC and ITR access modes	3022
E4.4	Flow control of the DCC and ITR registers	3026
E4.5	Synchronization of DCC and ITR accesses	3029
E4.6	Interrupt-driven use of the DCC	3034
E4.7	Pseudocode description of the operation of the DCC and ITR registers	3035
Chapter E5	The Embedded Cross-Trigger Interface	
E5.1	About the Embedded Cross-Trigger	3037
E5.2	Basic operation on the ECT	3039
E5.3	Cross-triggers on a PE in an Arm R-profile implementation	3043
E5.4	Description and allocation of CTI triggers	3044
E5.5	CTI registers programmers' model	3047
E5.6	Examples.....	3048
Chapter E6	Debug Reset and Powerdown Support	
E6.1	About Debug over powerdown	3052
E6.2	Power domains and debug	3053
E6.3	Core power domain power states	3054
E6.4	Powerup request mechanism	3056
E6.5	Emulating low-power states	3057

E6.6	Debug OS Save and Restore sequences	3059
E6.7	Reset and debug	3064
Chapter E7	The PC Sample-based Profiling Extension	
E7.1	About the PC Sample-based Profiling Extension	3067
Chapter E8	About the External Debug Registers	
E8.1	Relationship between external debug and System registers	3070
E8.2	Endianness and supported access sizes.....	3071
E8.3	Synchronization of changes to the external debug registers	3072
E8.4	Memory-mapped accesses to the external debug interface	3076
E8.5	External debug interface register access permissions.....	3078
E8.6	External debug interface registers.....	3081
E8.7	Cross-trigger interface registers	3086
E8.8	External debug register resets	3088
Chapter E9	External Debug Register Descriptions	
E9.1	About the external debug registers.....	3091
E9.2	External debug registers	3092
E9.3	External trace registers	3200
E9.4	Cross-Trigger Interface registers	3201
Part F	Memory-mapped Components of the Arm Architecture	
Chapter F1	Requirements for Memory-mapped Components	
F1.1	Supported access sizes and Endianness	3267
F1.2	Synchronization of memory-mapped registers	3269
F1.3	Access requirements for reserved and unallocated registers	3271
Chapter F2	System Level Implementation of the Generic Timer	
F2.1	About the Generic Timer specification	3273
F2.2	Memory-mapped counter module	3275
F2.3	Memory-mapped timer components	3279
Chapter F3	Recommended External Interface to the Performance Monitors	
F3.1	About the external interface to the Performance Monitors registers	3284
Chapter F4	External System Control Register Descriptions	
F4.1	About the external system control register descriptions	3289
F4.2	External Performance Monitors registers summary	3290
F4.3	Performance Monitors external register descriptions	3292

F4.4	Generic Timer memory-mapped registers overview.....	3360
F4.5	Generic Timer memory-mapped register descriptions	3361

Part G Architectural Pseudocode

Chapter G1 R-profile Architecture Pseudocode

G1.1	Pseudocode for AArch32 operation	3409
G1.2	Shared pseudocode.....	3491

Part H Appendixes

Appendix H1 Architectural Constraints on UNPREDICTABLE Behaviors

H1.1	AArch32 CONSTRAINED UNPREDICTABLE behaviors.....	3648
------	--	------

Appendix H2 Recommendations for Reporting Memory Attributes on an Interconnect

H2.1	Arm recommendations for reporting memory attributes on an interconnect	3666
------	--	------

Appendix H3 Legacy Instruction Syntax for AArch32 Instruction Sets

H3.1	Legacy Instruction Syntax.....	3668
------	--------------------------------	------

Appendix H4 Arm Pseudocode Definition

H4.1	About the Arm pseudocode.....	3675
H4.2	Pseudocode for instruction descriptions.....	3676

Appendix H5 Registers Index

H5.1	Alphabetical index of AArch32 registers and System instructions	3678
H5.2	Functional index of AArch32 registers and System instructions	3685
H5.3	Alphabetical index of memory-mapped registers	3692
H5.4	Functional index of memory-mapped registers.....	3699

Glossary

Preface

This preface introduces the *Arm Architecture Reference Manual, for R-profile AArch32 architecture*. It contains the following sections:

- [About this Manual.](#)
- [Using this Manual.](#)
- [Conventions.](#)
- [Additional reading.](#)
- [Feedback.](#)

About this Manual

This Manual describes the *Arm[®] architecture v8, Armv8 for R-profile AArch32 architecture*. It supersedes and replaces DDI0568 *Arm[®] Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch32 architecture profile*. DDI0568 is deprecated.

This issue of the Manual contains many improvements and corrections.

The features implemented by the R-profile architecture are described in [R-profile Architecture Extensions](#). The following sections contain references to features that are not implemented in Armv8-R-compliant Processing Elements (PEs). These references can be ignored when interpreting the architecture for Armv8-R:

- The [Ordering requirements defined by the formal concurrency model](#). This is to ensure that the memory model is conformant with the [formal concurrency model](#).
- The [AArch32 System Register Descriptions](#).
- The [AArch32 Instruction Sets](#).
- The [PMU Event Descriptions](#).

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

- Some diagrams in the register descriptions chapter have long field names that split over several lines. These are not wrong, and the descriptions list the field name correctly.
- [Arm Pseudocode Definition](#) requires further review and update. Since this appendix is informative, rather than being part of the architecture specification, this does not affect the quality status of this release.

The architecture describes the operation of an Armv8-R AArch32 architecture [Processing element \(PE\)](#), and this Manual includes descriptions of:

- The Execution state AArch32.
- The T32 and A32 instruction sets, which are compatible with earlier versions of the Arm architecture.
- The states that determine how a PE operates, including the current Exception level and Security state, and the PE mode.
- The Exception model.
- The memory model, that defines memory ordering and memory management. This Manual covers the Arm R-profile architecture profile that defines a *Protected Memory System Architecture* (PMSA).
- The programmers' model, and its interfaces to System registers that control most PE and memory system features, and provide status information.
- The Advanced SIMD and floating-point instructions, which provide high-performance:
 - Single-precision, half-precision, and double-precision floating-point operations.
 - Conversions between double-precision, single-precision, and half-precision floating-point values.
 - Integer, single-precision floating-point, and half-precision floating-point vector operations in all instruction sets.
- The Debug architecture, which provides software access to debug features.

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

This Manual is organized into parts:

Part A Provides an introduction to the Arm architecture, and an overview of the AArch32 Execution state.

Part B Describes the application level view of the AArch32 Execution state, meaning the view from the EL0. It describes the application level view of the programmers' model and the memory model.

———— **Note** —————

In AArch32 state, execution at EL0 is execution in User mode.

Part C Describes the T32 and A32 instruction sets, which are available in the AArch32 Execution state. These instruction sets are backwards-compatible with earlier versions of the Arm architecture. This part describes the precise effects of each instruction when executed in User mode, described as *unprivileged* execution or execution at EL0, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Note

User mode is the only mode where software execution is unprivileged.

Part D Describes the system level view of the AArch32 Execution state, which is generally compatible with earlier versions of the Arm architecture. This part includes details of the System registers, most of which are not accessible from EL0, and the instruction interface to those registers. It also describes the system level view of the programmers' model and the memory model.

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

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

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

Part H, Appendixes Provide additional information. Some appendixes give information that is not part of the A-profile architectural requirements. The cover page of each appendix indicates its status.

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

Note

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

Using this Manual

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

Part A, Introduction and Architecture Overview

Part A gives an overview of the Armv8-R for AArch32 architecture profile. It introduces the terminology used to describe the architecture. It contains the following chapters:

Introduction to the Arm Architecture

Read this for an introduction to the Arm architecture.

R-profile AArch32 Architecture Extensions

Read this for an introduction to the R-profile AArch32 architecture extensions and features.

Part B, The AArch32 Application Level Architecture

Part B describes the AArch32 state application level view of the architecture. It contains the following chapters:

The AArch32 Application Level Programmers' Model

Read this for an application level description of the programmers' model for software executing in AArch32 state. It describes execution at EL0 when EL0 is using AArch32 state.

The AArch32 Application Level Memory Model

Read this for an application level description of the memory model for software executing in AArch32 state. It describes the memory model for execution in EL0 when EL0 is using AArch32 state. It includes information about Arm memory types, attributes, and memory access controls.

Part C, The AArch32 Instruction Sets

Part C describes the T32 and A32 instruction sets, which are used in AArch32 state. It contains the following chapters:

About the T32 and A32 Instruction Descriptions

Read this to understand the format of the T32 and A32 instruction descriptions.

The AArch32 Instruction Sets Overview

Read this for an overview of the T32 and A32 instruction sets.

T32 Instruction Set Encoding

Read this for a description of the T32 instruction set encoding. This includes the T32 encoding of the Advanced SIMD and floating-point instructions.

A32 Instruction Set Encoding

Read this for a description of the A32 instruction set encoding. This includes the A32 encoding of the Advanced SIMD and floating-point instructions.

T32 and A32 Base Instruction Set Instruction Descriptions

Read this for a description of each of the T32 and A32 base instructions.

T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

Read this for a description of each of the T32 and A32 Advanced SIMD and floating-point instructions.

Part D, The AArch32 System Level Architecture

Part D describes the AArch32 state system level view of the architecture. It contains the following chapters:

The AArch32 System Level Programmers' Model

Read this for a description of the AArch32 state system level view of the programmers' model for execution in an Exception level that is using AArch32.

AArch32 Self-hosted Debug

Read this for an introduction to, and a description of, self-hosted debug in AArch32 state.

AArch32 Self-hosted Trace

Read this for an introduction to, and a description of, self-hosted trace in AArch32 state.

The AArch32 System Level Memory Model

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

The Generic Timer in AArch32 state

Read this for a description of the AArch32 view of an implementation of the Arm Generic Timer.

AArch32 System Register Encoding

Read this for a description of the encoding of the AArch32 System registers, including the System instructions that are part of the AArch32 System registers encoding space.

AArch32 System Register Descriptions

Read this for a description of each of the AArch32 System registers.

Part E, External Debug

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

About External Debug

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

Debug State

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

Halting Debug Events

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

The Debug Communication Channel and Instruction Transfer Register

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

The Embedded Cross-Trigger Interface

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

Debug Reset and Powerdown Support

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

The PC Sample-based Profiling Extension

Read this for a description of the PC Sample-based Profiling Extension that is an OPTIONAL extension.

About the External Debug Registers

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

External Debug Register Descriptions

Read this for a description of each external debug register.

Part F, Memory-mapped Components of the Arm architecture

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

Requirements for Memory-mapped Components

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

System Level Implementation of the Generic Timer

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

Recommended External Interface to the Performance Monitors

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

External System Control Register Descriptions

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

Part G, Architectural Pseudocode

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

R-profile Architecture Pseudocode

Read this for the pseudocode definitions that describe various features of the Arm architecture, for operation in AArch32 state.

Part H, Appendixes

Note

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

This Manual contains the following appendixes:

Architectural Constraints on UNPREDICTABLE Behaviors

Read this for a description of the architecturally-required constraints on UNPREDICTABLE behaviors in the Arm architecture, including AArch32 behaviors that were UNPREDICTABLE in previous versions of the architecture.

Arm Pseudocode Definition

Read this for definitions of the Arm pseudocode.

Registers Index

Read this for an alphabetic and functional index of AArch32 registers, and memory-mapped registers.

Glossary

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

Note

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

Conventions

The following sections describe conventions that this book can use:

- [Typographic conventions.](#)
- [Rules-based writing.](#)
- [Signals.](#)
- [Numbers.](#)
- [Pseudocode descriptions.](#)
- [Assembler syntax descriptions.](#)

Typographic conventions

The typographical conventions are:

<i>italic</i>	Introduces special terminology, and denotes citations.
bold	Denotes signal names, and is used for terms in descriptive lists, where appropriate.
<code>monospace</code>	Used for assembler syntax descriptions, pseudocode, and source code examples. Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

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

Colored text	Indicates a link. This can be: • A URL, for example https://developer.arm.com. • A link, to a chapter or appendix, or to a glossary entry, or to the section of the Manual that defines the colored term, for example Simple sequential execution or SCTLR.
---------------------	---

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

Optional items

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

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

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

Sets of items

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

Notes	Notes are formatted as:
--------------	-------------------------

Note

This is a Note.

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

Rules-based writing

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

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

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

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

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

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

Content item identifiers

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

Content item rendering

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

Content item classes

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

Rule

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

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label R.

Information

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

An Information statement is rendered with the label I.

Software usage

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

A Software usage statement is rendered with the label S.

Declaration

A Declaration statement introduces concepts or terminology.

A Declaration does not describe behavior.

A Declaration is rendered with the label D.

Signals

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

- Signal level** The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:
- HIGH for active-HIGH signals.
 - LOW for active-LOW signals.
- Lowercase n** At the start or end of a signal name denotes an active-LOW signal.

Numbers

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

Pseudocode descriptions

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

Assembler syntax descriptions

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

Additional reading

This section lists relevant publications from Arm and third parties.

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

Arm publications

- *AMBA[®] 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1* (ARM IHI 0032).
- *Arm[®] Architecture Reference Manual for A-profile architecture*, (ARM DDI 0487).
- *Arm[®] Architecture Reference Manual for R-profile AArch64 architecture*, (ARM DDI 0628).
- *Arm[®] Architecture Specification Language Reference* (ARM DDI 0626).
- *Arm[®] Architecture Specification Language Readers' Guide* (ARM 111069).
- *ARM[®] Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *Arm[®] Debug Interface Architecture Specification, ADIv6.0* (ARM IHI 0074).
- *Arm[®] Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *Arm[®] Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
- *Arm[®] Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).
- *ARM[®] CoreSight[™] SoC Technical Reference Manual* (ARM DDI 0480).
- *Arm[®] CoreSight[™] Architecture Specification* (ARM IHI 0029).

Other publications

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

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

Feedback

Arm welcomes feedback on its documentation.

Feedback on this Manual

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

As part of the ticket, include:

- The title, Arm® Architecture Reference Manual, for R-profile AArch32 architecture.
- The number, DDI 111353 A.a.
- The section name to which your comments refer.
- The page number(s) to which your comments refer.
- The rule identifier(s) to which your comments refer, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader. Search performance may be significantly improved by increasing the Fast Find Maximum Cache Size under Search in Preferences.

If using Google Chrome™ or Microsoft Edge™, load time is significantly improved since the April 2025 release of these browsers.

Users may find, on some systems, that PDF viewers such as Okular or Foxit™ give good performance.

Inclusive terminology commitment

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

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

Part A **Arm Architecture Introduction and Overview**

Chapter A1

Introduction to the Arm Architecture

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

- [About the Arm architecture.](#)
- [Architecture profiles.](#)
- [Arm architectural concepts.](#)
- [Supported data types.](#)
- [Floating-point support.](#)
- [The Arm memory model.](#)

A1.1 About the Arm architecture

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

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

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

The Arm architecture includes definitions of:

- An associated debug architecture, see:
 - [AArch32 Self-hosted Debug](#).
 - External Debug.
- Associated trace architectures that define trace units that implementers can implement with the associated processor hardware. For more information, see *The Arm® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).

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

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

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

This document defines version 8 of the R-profile AArch32 architecture.

See [Architecture profiles](#) for more information.

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

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

The Arm architecture supports a 32-bit Execution state, AArch32, that is compatible with previous versions of the Arm architecture.

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

Note

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

The Arm architecture supports Scalar floating-point instructions that operate on registers in the SIMD&FP register file.

Note

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

A1.2 Architecture profiles

The Arm architecture has evolved significantly since its introduction, and Arm continues to develop it. Nine major versions of the architecture have been defined to date, denoted by the version numbers 1 to 9. The R-profile AArch32 architecture is based on the Armv8 architectural features.

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

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

AArch32 Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

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

See sections [Execution state](#) and [The instruction sets](#) for more information.

Arm defines three architecture profiles:

- A** Application profile:
- Supports a *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).
-
- Note**
An Armv8-A implementation can be called an AArchv8-A implementation and an Armv9-A implementation can be called an AArchv9-A implementation.
-
- Supports the A64, A32, and T32 instruction sets.
- R** Real-time profile, described in this Manual:
- Supports a *Protected Memory System Architecture* (PMSA) based on a *Memory Protection Unit* (MPU).
 - Supports an optional VMSA based on an MMU in the R-profile AArch64 architecture.
 - Supports the A64 instruction set if the R-profile AArch64 architecture is implemented, or the A32 and T32 instruction sets if the R-profile AArch32 architecture is implemented.
- M** Microcontroller profile:
- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
 - Supports a PMSA based on an MPU.
 - Supports a variant of the T32 instruction set.

This Manual describes only Armv8-R AArch32. For information about the A, R AArch64 and M architecture profiles, and earlier Arm architecture versions, see:

- The *Arm[®] Architecture Reference Manual, for A-profile architecture*.
- The *Arm[®] Architecture Reference Manual, for Armv8-R AArch64 architecture profile*.
- The *ARM[®] Architecture Reference Manual ARMv7-A and ARMv7-R edition*.
- The *Armv8-M Architecture Reference Manual*.
- The *Armv7-M Architecture Reference Manual*.
- The *ARMv6-M Architecture Reference Manual*.

A1.3 Arm architectural concepts

This section introduces architectural concepts and the associated terminology.

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

- [Execution state](#).
- [The instruction sets](#).
- [System registers](#).
- [Arm Debug](#).

A1.3.1 Execution state

The Execution state defines the PE execution environment, including:

- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
 - The Exception model.
 - The programmers' model.

The Execution state is:

- AArch32** The 32-bit Execution state. This Execution state:
- Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and *Link Register* (LR). The LR is used as both an ELR and a procedure link register. Some of these registers have multiple *banked* instances for use in different PE *modes*.
 - Provides a single ELR, for exception returns from Hyp mode.
 - Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
 - Provides two instruction sets, T32 and A32. For more information, see [The instruction sets](#).
 - Supports the Armv7-A Exception model, based on *PE modes*, and maps this onto the Armv8 Exception model, that is based on the Exception levels.
 - Provides support for 32-bit virtual addressing.
 - Defines a number of *Process state* (**PSTATE**) elements that hold PE state. The T32 and A32 instruction sets include instructions that operate directly on various **PSTATE** elements, and instructions that access **PSTATE** by using the *Application Program Status Register* (APSR) or the *Current Program Status Register* (CPSR).

A1.3.2 Security state

The architecture defines one *Security state*, Non-secure state.

A1.3.3 The instruction sets

The possible instruction sets depend on the Execution state:

- AArch32** AArch32 state supports the following instruction sets:
- | | |
|------------|---|
| A32 | This is a fixed-length instruction set that uses 32-bit instruction encodings. |
| T32 | This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings. |

In previous documentation, these instruction sets were called the ARM and Thumb[®] instruction sets. Armv8 and Armv9 extend each of these instruction sets. In AArch32 state, the Instruction set state determines the instruction set that the PE executes.

For information on the T32 and A32 instruction sets, see [The AArch32 Instruction Sets Overview](#).

The instruction sets support SIMD and scalar floating-point instructions. See [Floating-point support](#).

A1.3.4 System registers

System registers provide control and status information of architected features.

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

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

For information about Exception levels, see Exception levels.

The System registers comprise:

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

For information about the AArch32 System registers, see [AArch32 System Register Descriptions](#).

A1.3.4.1 The Arm Generic Interrupt Controller System registers

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

- [About the GIC System registers](#), for more information about the AArch32 GIC System registers.

This section gives only short overviews of the GIC System registers. For more information, including descriptions of the registers, see the *Arm[®] Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).

Note

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

A1.3.5 Arm Debug

Armv8 and later architectures support the following:

Self-hosted debug

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

External debug

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

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

For more information about self-hosted debug, see [AArch32 Self-hosted Debug](#).

For more information about external debug, see [External Debug](#).

A1.4 Supported data types

The Arm architecture supports the following integer data types:

Byte	8 bits.
Halfword	16 bits.
Word	32 bits.
Doubleword	64 bits.
Quadword	128 bits.

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

- Half-precision. See [Half-precision floating-point formats](#).
- Single-precision. See [Single-precision floating-point format](#).
- Double-precision. See [Double-precision floating-point format](#).

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

The architecture provides the following:

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

In AArch32 state:

- The general-purpose register file contains 32-bit registers:
 - Two 32-bit registers can support a doubleword.
 - Vector formatting is supported, see [Figure A1-3](#).
- The SIMD&FP register file contains 64-bit registers:
 - AArch32 state does not support quadword integer or floating-point data types.

Note

Two consecutive 64-bit registers can be used as a 128-bit register.

For more information on the register files in AArch32 state, see [The general-purpose registers, and the PC, in AArch32 state](#).

A1.4.1 Advanced SIMD vector formats

In an implementation that includes the Advanced SIMD instructions that operate on the SIMD&FP register file, a register can hold one or more packed elements, all of the same size and type. In AArch32 state, the combination of a register and a data type describes a vector of elements, where the number of elements in the vector is implied by the size of the data type and the size of the register.

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

For more information on the Advanced SIMD and floating-point registers in AArch32 state, see [The SIMD and floating-point register file](#).

A1.4.1.1 Advanced SIMD vector formats in AArch32 state

Table A1-1 shows the available formats. Each instruction description specifies the data types that the instruction supports.

Table A1-1 Advanced SIMD data types in AArch32 state

Data type specifier	Meaning
.<size>	Any element of <size> bits
.F<size>	Floating-point number of <size> bits
.I<size>	Signed or unsigned integer of <size> bits
.P<size>	Polynomial over {0, 1} of degree less than <size>
.S<size>	Signed integer of <size> bits
.U<size>	Unsigned integer of <size> bits

Polynomial arithmetic over {0, 1} describes the polynomial data type.

The .F16 data type is the half-precision data type selected by the FPSCR.AHP bit, see [Half-precision floating-point formats](#).

The .F32 data type is the Arm standard single-precision floating-point data type, see [Single-precision floating-point format](#).

The instruction definitions use a data type specifier to define the data types appropriate to the operation. Figure A1-1 shows the hierarchy of the Advanced SIMD data types.

.8	.I8	.S8
		.U8
	.P8	
	-	
.16	.I16	.S16
		.U16
	.P16 †	
	.F16	
	.BF16	
.32	.I32	.S32
		.U32
	-	
	.F32	
.64	.I64	.S64
		.U64
	.P64 ‡	
	-	

† Output format only. See VMULL instruction description.

‡ Available only if the Cryptographic Extension is implemented. See VMULL instruction description.

Figure A1-1 Advanced SIMD data type hierarchy in AArch32 state

For example, a multiply instruction must distinguish between integer and floating-point data types.

An integer multiply instruction that generates a double-width (long) result must specify the input data types as signed or unsigned. However, some integer multiply instructions use modulo arithmetic, and therefore do not have to distinguish

between signed and unsigned inputs.

Figure A1-2 shows the Advanced SIMD vectors in AArch32 state.

Note

In AArch32 state, a pair of even and following odd numbered doubleword registers can be concatenated and treated as a single quadword register.

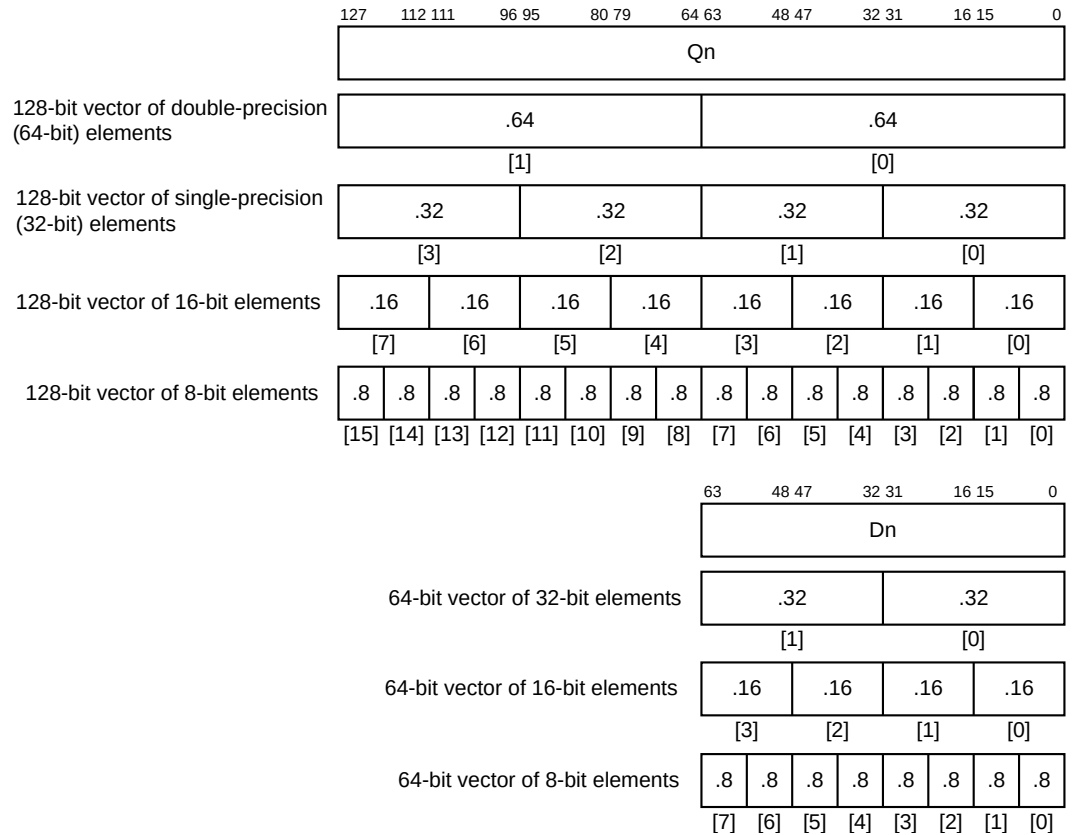


Figure A1-2 Advanced SIMD vectors in AArch32 state

The AArch32 general-purpose registers support vectors formats for use by the SIMD instructions in the Base instruction set. Figure A1-3 shows these formats, that means that a general-purpose register can be treated as either 2 halfwords or 4 bytes.

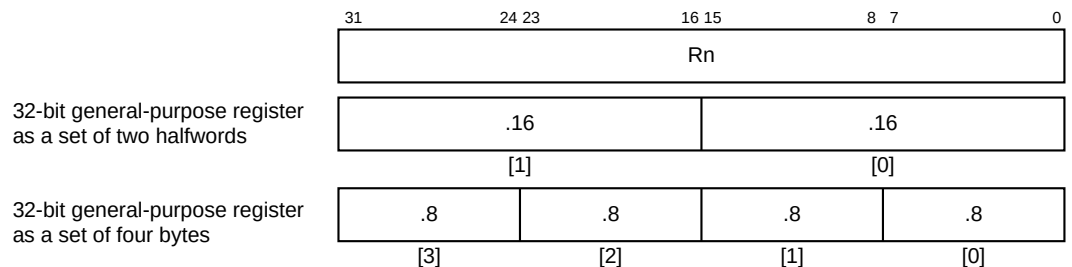


Figure A1-3 Vector formatting in AArch32 state

A1.4.2 Half-precision floating-point formats

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

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

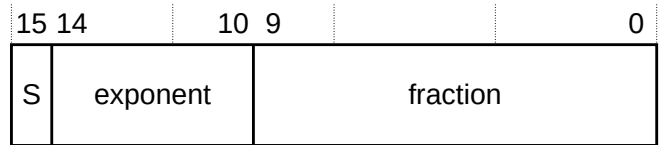
- Arm *alternative half-precision* format.

Both formats can be used for conversions to and from other floating-point formats.

[FPSCR.AHP](#) controls the format in AArch32 state.

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

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



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

0 < exponent < 0x1F

The value is a normalized number and is equal to:

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

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

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

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

exponent == 0

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

fraction == 0

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

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

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

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

Half-precision denormalized numbers are not flushed to zero by default. For details, see [Flushing denormalized numbers to zero](#).

exponent == 0x1F

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

IEEE half-precision

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

fraction == 0

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

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

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

fraction := 0

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

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

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

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

Alternative half-precision

The value is a normalized number and is equal to:

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

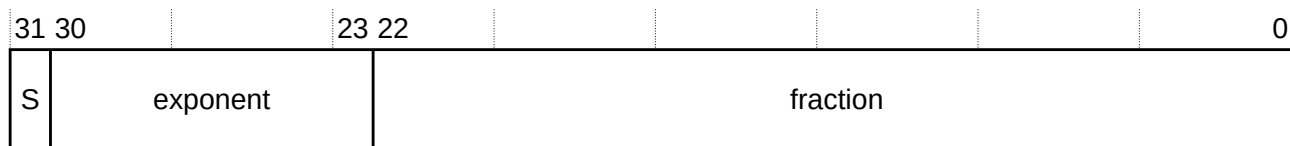
The maximum positive normalized number is $(2 \cdot 2^{-10}) \times 2^{16}$ or 131008.

A1.4.3 Single-precision floating-point format

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

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

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



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

$$0 < \text{exponent} < 0_{\text{xFF}}$$

The value is a normalized number and is equal to:

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

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

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

exponent == 0

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

fraction == 0

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

+0 When $S=0$.

-0 When $S=1$.

These usually behave identically. In particular, the result is *equal* if +0 and -0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:

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

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

Denormalized numbers are always flushed to zero in Advanced SIMD processing in AArch32 state. are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see [Flushing denormalized numbers to zero](#).

exponent == 0xFF

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

fraction == 0

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

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

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

fraction != 0

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

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

bit[22] == 0

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

bit[22] == 1

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

For details of the *default NaN*, see [Default NaN](#).

Note

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

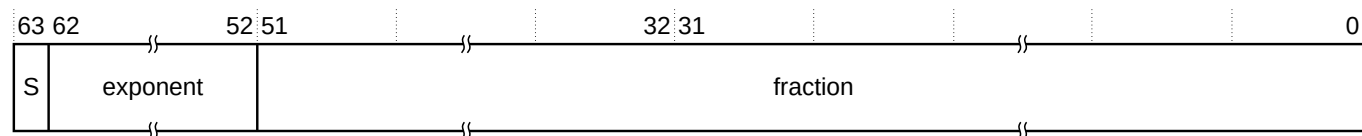
A1.4.4 Double-precision floating-point format

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

Double-precision floating-point is supported only by floating-point instructions.

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

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



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

$$0 < \text{exponent} < 0x7F$$

The value is a normalized number and is equal to:

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

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

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

exponent == 0

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

fraction == 0

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

$$+0 \quad \text{when } S=0.$$

-0 when $S=1$.

fraction **!= 0**

The value is a denormalized number and is equal to:

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

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

Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details, see [Flushing denormalized numbers to zero](#).

```
exponent == 0x7FF
```

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

fraction == 0

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

+infinity When $S=0$.

-infinity When $S=1$.

fraction **!= 0**

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

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

bit[51] == 0

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

bit[51] == 1

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

For details of the *default NaN*, see [Default NaN](#).

Note

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

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

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

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

These instructions can cause the following floating-point exceptions:

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

Inexact When the numeric result differs from the input value.

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

Note

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

For more information, see [Floating-point exceptions and exception traps](#).

A1.5 Floating-point support

The Armv8 architecture includes Scalar floating-point instructions that operate on the lowest numbered element of the SIMD&FP registers.

Armv8-R AArch32 also supports single-precision floating-point implementations without Advanced SIMD.

If an implementation supports only single-precision floating-point operations and does not support Advanced SIMD, then it must implement only 16 double-precision registers, D0-D15, that is, it must be a D16 implementation.

For more information about the permitted floating-point and Advanced SIMD support, see [MVFR0](#), [MVFR1](#) and [MVFR2](#).

The definitions of [HCPTR.TASE](#), [NSACR](#), and [CPACR](#) apply for all implementations, including D16.

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

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

The architecture supports the IEEE 754 single-precision for scalar data processing. For many floating-point instructions, there are configurable behaviors, such as:

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

Floating-point computation using AArch32 Advanced SIMD instructions remains unchanged from Armv7. T32 and A32 floating-point instructions always use Arm standard floating-point arithmetic and performs IEEE 754 floating-point arithmetic with the following restrictions:

- Denormalized numbers are flushed to zero, see [Flushing denormalized numbers to zero](#).
- Only default NaNs are supported, see [Default NaN](#).
- The Round to Nearest rounding mode is used.
- Untrapped floating-point exception handling is used for all floating-point exceptions.

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

For more information, see [Floating-point exceptions and exception traps](#).

There is a single Floating-Point Status and Control Register, [FPSCR](#).

The [FPSCR](#).{N,Z,C,V} condition flags are updated by floating-point comparison operations.

Note

For additional details of AArch32 state floating-point support, see [Advanced SIMD and floating-point support](#).

The remainder of this section contains:

- [Instruction support](#).
- [Floating-point standards, and terminology](#).
- [Arm standard floating-point input and output values](#).
- [Flushing denormalized numbers to zero](#).
- [NaN handling and the Default NaN](#).
- [Rounding](#).
- [Floating-point exceptions and exception traps](#).

A1.5.1 Instruction support

The floating-point instructions support:

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

Note

Single elements are also referred to as scalar elements.

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

For more information on floating-point instructions in AArch32 state, see [The AArch32 Instruction Sets Overview](#).

A1.5.2 Floating-point standards, and terminology

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

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

[Table A1-2](#) shows how the terminology in this Manual differs from that used in IEEE 754-2008.

Table A1-2 Floating-point terminology

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

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

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

The Arm architecture provides full IEEE 754 floating-point arithmetic support. In AArch32 state, floating-point operations performed using Advanced SIMD instructions are limited to *Arm standard floating-point operation*, regardless of the selected rounding mode in the [FPSCR](#).

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

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

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

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

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

A1.5.4 Flushing denormalized numbers to zero

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

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

`MinNorm` is defined as follows:

- For half-precision numbers, `MinNorm` is 2^{-14} .
- For single-precision numbers, `MinNorm` is 2^{-126} .
- For double-precision numbers, `MinNorm` is 2^{-1022} .

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

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

A1.5.4.1 Flushing denormalized inputs to zero

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

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

If `FPSCR.FZ` == 0, for all inputs, `FPSCR.FZ` does not cause denormalized inputs to be flushed to zero, although other factors might cause denormalized inputs to be flushed to zero.

A1.5.4.2 Flushing denormalized outputs to zero

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

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

If $\text{FPSCR.FZ} = 0$, then unless otherwise specified, for all outputs, FPSCR.FZ does not cause denormalized outputs to be flushed to zero, although other factors might cause denormalized outputs to be flushed to zero.

If $\text{FPSCR.FZ} = 1$, for all outputs, denormalized outputs are flushed to zero.

A1.5.5 NaN handling and the default NaN

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

The Arm architecture additionally defines the following in [Table A1-3](#):

- A *default NaN*, compliant with the IEEE 754 standard.

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

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

A1.5.5.1 Default NaN

A default NaN is encoded as described in [Table A1-3](#).

Table A1-3 Default NaN encodings

	Half-precision, IEEE Format	Single- precision	Double- precision
Exponent	0x1F	0xFF	0x7FF
Fraction	Bit[9] = '1', bits[8:0] = 0	Bit[22] = '1', bits[21:0] = 0	Bit[51] = '1', bits[50:0] = 0

When $\text{FPSCR.DN} = 1$, if any input to a floating-point operation is a NaN, the output is a default NaN.

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

A1.5.5.2 NaN handling

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

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

A1.5.5.3 NaN propagation

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

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

- If the input NaN is a quiet NaN, the output NaN is derived from the input NaN.

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

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

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

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

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

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

A1.5.6 Rounding

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

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

`FPSCR.RMode` can select one of:

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

[Round to Nearest with Ties to Away](#) rounding behaviors are not selected by `FPSCR.RMode`, but are used by some instructions.

A1.5.6.1 Round to Nearest (RN) mode

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

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

A1.5.6.2 Round towards Plus Infinity (RP) mode

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

A1.5.6.3 Round towards Minus Infinity (RM) mode

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

A1.5.6.4 Round towards Zero (RZ) mode

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

A1.5.6.5 Round to Nearest with Ties to Away

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

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

A1.5.7 Floating-point exceptions and exception traps

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

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

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

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

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

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

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

A1.5.7.1 Inexact exceptions

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

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

A1.5.7.2 Underflow exceptions

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

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

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

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

A1.5.7.3 Overflow exceptions

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

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

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

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

A1.5.7.4 Divide by Zero exceptions

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

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

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

A1.5.7.5 Invalid Operation exceptions

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

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

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

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

A1.5.7.6 Handling floating-point exceptions

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

The `FPProcessException()` pseudocode function describes the handling of trapped floating-point exceptions generated in AArch32 state.

A1.5.7.7 Combinations of floating-point exceptions

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

- Overflow with Inexact.
- Underflow with Inexact.

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

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

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

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

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

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

A1.6 The Arm memory model

The Arm memory model supports:

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

AArch32 state VA supports 32-bit addressing, with the Protected Memory System Architecture defining protection regions arbitrated by Memory Protection Units at EL1 and EL2.

For more information, see:

For execution in AArch32 state

- [The AArch32 Application Level Memory Model.](#)
- [The AArch32 System Level Memory Model.](#)
- [The Protected Memory System Architecture.](#)

Chapter A2

R-profile Architecture Extensions

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

- [About the Armv8-R AArch32 architecture extensions.](#)
- [Armv8-R AArch32 architecture extensions.](#)

A2.1 About the Armv8-R AArch32 architecture extensions

An architecture extension defines a set of features that, when implemented, may make an implementation compliant with the extension.

Each feature description includes:

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

A2.2 Armv8-R AArch32 architecture extensions

A2.2.1 The Armv8.0 architecture extension

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

An Armv8.0 compliant implementation can additionally include:

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

FEAT_AA32, PE supports AArch32.

This feature is mandatory.

FEAT_AA32EL0, Support for AArch32 at EL0

This feature is mandatory.

FEAT_AA32EL1, Support for AArch32 at EL1

This feature is mandatory.

FEAT_AA32EL2, Support for AArch32 at EL2

This feature is mandatory.

FEAT_AES, Advanced SIMD AES instructions

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

If FEAT_AES is implemented, then [FEAT_Crypto](#) is implemented.

FEAT_AdvSIMD, Advanced SIMD Extension

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

The following field identifies the presence of FEAT_AdvSIMD:

- [EDPFR](#).AdvSIMD.

For more information, see:

- [Supported data types](#).
- A64 Advanced SIMD and Floating-point Instruction Descriptions.
- [Advanced SIMD and floating-point instructions](#).
- [T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions](#).

FEAT_BigEnd, Support for big-endian at EL1 and above

If FEAT_BigEnd is implemented, then [FEAT_BigEndEL0](#) is implemented.

If [FEAT_MixedEnd](#) is implemented, then FEAT_BigEnd is implemented.

FEAT_BigEndEL0, Support for big-endian at EL0

If [FEAT_MixedEndEL0](#) is implemented, then FEAT_BigEndEL0 is implemented.

FEAT_CRC32, CRC32 instructions

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

FEAT_CTI_PIDR, Support for CoreSight compliance for external CTI registers

FEAT_CTI_PIDR provides registers for compliance with the CoreSight architecture.

This feature is required for compliance with the CoreSight architecture.

FEAT_Crypto, Cryptographic Extension

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

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

In an implementation that supports both AArch64 state and AArch32 state, FEAT_AES, FEAT_PMULL, FEAT_SHA1 and FEAT_SHA256 provide the same functionality in both states.

If FEAT_Crypto is implemented, then [FEAT_AES](#) or [FEAT_SHA1](#) is implemented.

For more information, see:

- The Cryptographic Extension in AArch64 state.
- [The Cryptographic Extension in AArch32 state.](#)

FEAT_DoubleLock, Double Lock

FEAT_DoubleLock is the mnemonic used for the OS Double Lock.

The following field identifies the presence of FEAT_DoubleLock:

- [DBGDEVID](#).DoubleLock.

FEAT_ED_PIDR, Support for CoreSight compliance for External Debug registers

FEAT_ED_PIDR provides registers for compliance with the CoreSight architecture.

This feature is required for compliance with the CoreSight architecture.

For more information, see Management registers and CoreSight compliance.

FEAT_EL0, Support for execution at EL0

This feature is mandatory.

FEAT_EL1, Support for execution at EL1

This feature is mandatory.

FEAT_EL2, Support for execution at EL2

This feature is mandatory.

FEAT_ETMv4, Embedded Trace Macrocell version 4

FEAT_ETMv4 indicates support for the Embedded Trace Macrocell architecture ETMv4.

If FEAT_ETMv4 is implemented, then [FEAT_TRC_SR](#) or [FEAT_TRC_EXT](#) is implemented.

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

FEAT_FP, Floating Point extensions

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

The following field identifies the presence of FEAT_FP:

- [EDPFR.FP](#).

For more information, see:

- [Supported data types](#).
- [Floating-point support](#).
- [Advanced SIMD and floating-point instructions](#).
- [T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions](#).

FEAT_IVIPT, The IVIPT Extension

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

FEAT_LittleEnd, Support for little-endian at EL1 and above

If FEAT_LittleEnd is implemented, then [FEAT_LittleEndEL0](#) is implemented.

If [FEAT_MixedEnd](#) is implemented, then FEAT_LittleEnd is implemented.

FEAT_LittleEndEL0, Support for little-endian at EL0

If [FEAT_MixedEndEL0](#) is implemented, then FEAT_LittleEndEL0 is implemented.

FEAT_MixedEnd, Mixed-endian support

FEAT_MixedEnd provides support for mixed-endian configuration.

If FEAT_MixedEnd is implemented, then [FEAT_MixedEndEL0](#) is implemented.

For more information, see:

- [Mixed-endian support in AArch32](#).

FEAT_MixedEndEL0, Mixed-endian support at EL0

FEAT_MixedEndEL0 provides support for mixed-endian at EL0.

For more information, see:

- [Mixed-endian support in AArch32](#).

FEAT_PCSRv8, PC Sample-based Profiling extension

FEAT_PCSRv8 introduces support for PC Sample-based Profiling Extension that provides coarse-grained, non-invasive profiling by an external debugger.

For more information, see [About the PC Sample-based Profiling Extension](#).

FEAT_PMSAv32, Protected Memory System Architecture v8-32

For more information, see [The AArch32 Protected Memory System Architecture](#)

FEAT_PMULL, Advanced SIMD PMULL instructions

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

If FEAT_PMULL is implemented, then [FEAT_AES](#) is implemented.

FEAT_PMU_PIDR, Support for CoreSight compliance for external PMU registers

FEAT_PMU_PIDR provides registers for compliance with the CoreSight architecture.

This feature is required for compliance with the CoreSight architecture.

If FEAT_PMU_PIDR is implemented, then [FEAT_PMUv3_EXT](#) is implemented.

For more information, see [Management registers and CoreSight compliance](#).

FEAT_PMUv3, Performance Monitors Extension, version 3

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

The following fields identify the presence of FEAT_PMUv3:

- [ID_DFR0](#).PerfMon.
- [EDDFR](#).PMUVer.

For more information, see [The Performance Monitors Extension](#).

FEAT_PMUv3_EXT, External interface to the Performance Monitors

FEAT_PMUv3_EXT indicates support for external access to the FEAT_PMUv3 registers.

If FEAT_PMUv3_EXT is implemented, then [FEAT_PMUv3](#) is implemented.

If FEAT_PMUv3_EXT is implemented, then [FEAT_PMUv3_EXT32](#) is implemented.

FEAT_PMUv3_EXT32, 32-bit external interface to the Performance Monitors

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

If FEAT_PMUv3_EXT32 is implemented, then [FEAT_PMUv3_EXT](#) is implemented.

If [FEAT_PMUv3_EXT](#) and [FEAT_PMUv3](#) are implemented, then FEAT_PMUv3_EXT32 is implemented.

For more information, see [Recommended External Interface to the Performance Monitors](#).

FEAT_SHA1, Advanced SIMD SHA1 instructions

FEAT_SHA1 implements the SHA1* instructions.

If FEAT_SHA1 is implemented, then [FEAT_Crypto](#) is implemented.

FEAT_SHA256, Advanced SIMD SHA256 instructions

FEAT_SHA256 implements the SHA256* instructions.

If FEAT_SHA256 is implemented, then [FEAT_SHA1](#) is implemented.

FEAT_TRC_EXT, Trace external registers

FEAT_TRC_EXT indicates support for external access to the ETMv4 registers.

If FEAT_TRC_EXT is implemented, then [FEAT_ETMv4](#) is implemented.

FEAT_TRC_SR, Trace System registers

FEAT_TRC_SR indicates support for System registers for ETMv4.

If FEAT_TRC_SR is implemented, then [FEAT_ETMv4](#) is implemented.

The following field identifies the presence of FEAT_TRC_SR:

- [ID_DFR0](#).CopTrc.

The AArch32 Application Level Architecture

Part B

Chapter B1

The AArch32 Application Level Programmers' Model

This chapter gives an Application level description of the programmers' model for software executing in AArch32 state. This means it describes execution in EL0. It contains the following sections:

- [About the Application level programmers' model](#)
- [The Application level programmers' model in AArch32 state](#)
- [Advanced SIMD and floating-point instructions](#)
- [About the AArch32 System register interface](#)
- [Exceptions](#)

B1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for the development of applications that execute in AArch32 state.

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

Depending on the implementation, the architecture supports multiple levels of execution privilege. These privilege levels are indicated by different *Exception levels* that number upwards from EL0, where EL0 corresponds to the lowest privilege level and is often described as *unprivileged*. The Application level programmers' model is the programmers' model for software executing at EL0. For more information, see [Arm architectural concepts](#).

System software determines the Exception level, and therefore the level of privilege, at which application software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged. This has the following effects:

- It means that the operating system can allocate system resources to an application in a unique or shared manner.
- It provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some System level understanding is helpful, and if appropriate it gives a reference to the System level description.

Application level software is generally unaware of any virtualization. For more information, see [The effect of implementing EL2 on the Exception model](#).

B1.2 The Application level programmers' model in AArch32 state

The following sections give more information about the application level programmers' model in AArch32 state:

- [Instruction sets, arithmetic operations, and register files.](#)
- [Core data types and arithmetic in AArch32 state.](#)
- [The general-purpose registers, and the PC, in AArch32 state.](#)
- [Process state, PSTATE.](#)
- [Jazelle support.](#)

B1.2.1 Instruction sets, arithmetic operations, and register files

The T32 and A32 instruction sets both provide a wide range of integer arithmetic and logical operations, that operate on a register file of sixteen 32-bit registers, that are comprised of the AArch32 general-purpose registers and the PC. As described in [The general-purpose registers, and the PC, in AArch32 state](#), these registers include the registers SP (R13) and LR (R14), which have specialized uses. [Core data types and arithmetic in AArch32 state](#) gives more information about these operations.

In addition, an implementation that implements the T32 and A32 instruction sets includes both:

- Scalar floating-point instructions.
- The Advanced SIMD vector instructions.

Floating-point and vector instructions operate on a separate common register file, described in [The SIMD and floating-point register file](#). [Advanced SIMD and floating-point instructions](#) gives more information about these instructions.

B1.2.2 Core data types and arithmetic in AArch32 state

When executing in AArch32 state, a PE supports the following data types in memory:

Byte	8 bits.
Halfword	16 bits.
Word	32 bits.
Doubleword	64 bits.

PE registers are 32 bits in size. The instruction sets provide instructions that use the following data types for data held in registers:

- 32-bit pointers.
- Unsigned or signed 32-bit integers.
- Unsigned 16-bit or 8-bit integers, held in zero-extended form.
- Signed 16-bit or 8-bit integers, held in sign-extended form.
- Two 16-bit integers packed into a register.
- Four 8-bit integers packed into a register.
- Unsigned or signed 64-bit integers held in two registers.

Load and store operations can transfer bytes, halfwords, or words to and from memory. Loads of bytes or halfwords zero-extend or sign-extend the data as it is loaded, as specified in the appropriate load instruction.

The instruction sets include load and store operations that transfer two or more words to and from memory. Software can load and store doublewords using these instructions.

Note

For information about the atomicity of memory accesses, see [Atomicity in the Arm architecture](#).

When any of the data types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to 2^N-1 , using normal binary format.

When any of these types is described as *signed*, the N-bit data value represents an integer in the range $-2^{(N-1)}$ to $+2^{(N-1)}-1$, using two's complement format.

The instructions that operate on packed halfwords or bytes include some multiply instructions that use only one of two halfwords, and SIMD instructions that perform parallel addition or subtraction on all of the halfwords or bytes.

Note

These SIMD instructions operate on values held in the general-purpose registers, and must not be confused with the Advanced SIMD instructions that operate on a separate register file that provides registers of up to 128 bits.

Direct instruction support for 64-bit integers is limited, and most 64-bit operations require sequences of two or more instructions to synthesize them.

B1.2.2.1 Integer arithmetic

The instruction set provides a wide range of operations on the values in registers, including bitwise logical operations, shifts, additions, subtractions, multiplications, and divisions. The pseudocode defines these operations either using ASL stdlib code or using ASL functions described in the *Arm[®] Architecture Specification Language Reference (ARM DDI 0626)*.

B1.2.2.1.1 Shift and rotate operations

The following types of shift and rotate operations are used in instructions:

Logical Shift Left

The `LSL()` stdlib function moves each bit of a bitstring left by a specified number of bits. Zeros are shifted in at the right end of the bitstring. Bits that are shifted off the left end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Logical Shift Right

The `LSR()` stdlib function moves each bit of a bitstring right by a specified number of bits. Zeros are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Arithmetic Shift Right

The `ASR()` stdlib function moves each bit of a bitstring right by a specified number of bits. Copies of the leftmost bit are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Rotate Right The `ROR()` stdlib function moves each bit of a bitstring right by a specified number of bits. Each bit that is shifted off the right end of the bitstring is re-introduced at the left end. The last bit shifted off the right end of the bitstring can be produced as a carry output.

Rotate Right with Extend

The `RRX()` pseudocode function moves each bit of a bitstring right by one bit. A carry input is shifted in at the left end of the bitstring. The bit shifted off the right end of the bitstring can be produced as a carry output.

B1.2.2.1.2 Pseudocode description of addition and subtraction

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand or operands otherwise.

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. When necessary, multi-word additions and subtractions can be synthesized from this status information. In pseudocode, the `AddWithCarry()` function provides an addition with a carry input and a set of output Condition flags including carry output and overflow:

An important property of the `AddWithCarry()` function is that if:

```
(result, nzcvc) = AddWithCarry{N}(x, NOT( y), carry_in)
```

Then:

- If `carry_in == '1'`, then `result == x-y` with:
 - `nzcvc[0] == '1'` if signed overflow occurred during the subtraction.
 - `nzcvc[1] == '1'` if unsigned borrow did not occur during the subtraction, that is, if $x \geq y$.
- If `carry_in == '0'`, then `result == x-y-1` with:
 - `nzcvc[0] == '1'` if signed overflow occurred during the subtraction.
 - `nzcvc[1] == '1'` if unsigned borrow did not occur during the subtraction, that is, if $x \geq y$.

Taken together, this means that the `carry_in` and `nzcvc[1]` output in `AddWithCarry{N}()` calls can act as NOT borrow flags for subtractions as well as carry flags for additions.

B1.2.2.1.3 Pseudocode description of saturation

Some instructions perform *saturating arithmetic*, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo 2^N . This is supported in pseudocode by:

- The `SignedSatQ{N}()` and `UnsignedSatQ{N}()` functions when an operation requires, in addition to the saturated result, a Boolean argument that indicates whether saturation occurred.
- The `SignedSat{N}()` and `UnsignedSat{N}()` functions when only the saturated result is required.

`SatQ{N}(i, unsigned)` returns either `UnsignedSatQ{N}(i)` or `SignedSatQ{N}(i)` depending on the value of its third argument, and `Sat(i, N, unsigned)` returns either `UnsignedSat{N}(i)` or `SignedSat{N}(i)` depending on the value of its third argument.

B1.2.3 The general-purpose registers, and the PC, in AArch32 state

In the AArch32 Application level view, a PE has:

- Fifteen general-purpose 32-bit registers, R0 to R14, of which R13 and R14 have alternative names reflecting how they are, or can be, used:
 - R13 is usually identified as SP.
 - R14 is usually identified as LR.
- The PC (*Program Counter*), which can be described as R15.

The specialized uses of the SP (R13), LR (R14), and PC (R15) are:

SP, the stack pointer

The PE uses SP as a pointer to the active stack.

In the T32 instruction set, some instructions cannot access SP. Instructions that can access SP can use SP as a general-purpose register.

The A32 instruction set provides more general access to SP, and it can be used as a general-purpose register.

Note

Using SP for any purpose other than as a stack pointer might break the requirements of operating systems, debuggers, and other software systems, causing them to malfunction.

Software can refer to SP as R13.

LR, the link register

The link register can be used to hold return link information, and some cases described in this manual require this use of the LR. When software does not require the LR for linking, it can use it for other purposes. Software can refer to LR as R14.

PC, the Program Counter

- When executing an A32 instruction, PC reads as the address of the current instruction plus 8.
- When executing a T32 instruction, PC reads as the address of the current instruction plus 4.
- Writing an address to PC causes a branch to that address.

Most T32 instructions cannot access PC.

The A32 instruction set provides more general access to the PC, and many A32 instructions can use the PC as a general-purpose register. However, Arm deprecates the use of PC for any purpose other than as the Program Counter. See [Writing to the PC](#) for more information.

Software can refer to PC as R15.

See [AArch32 general-purpose registers, the PC, and the Special-purpose registers](#) for the system level view of these registers.

Note

In general, Arm strongly recommends using the names SP, LR, and PC instead of R13, R14, and R15. However, sometimes it is simpler to use the R13-R15 names when referring to a group of registers. For example, it is simpler to refer to *registers R8 to R15*, rather than to *registers R8 to R12, the SP, LR, and PC*. These two descriptions of the group of registers have the same meaning.

B1.2.3.1 Writing to the PC

In the T32 and A32 instruction sets, many data-processing instructions can write to the PC. Writes to the PC are handled as follows:

- In T32 state, the following 16-bit T32 instruction encodings branch to the value written to the PC:
 - Encoding T2 of [ADD, ADDS \(register\)](#).
 - Encoding T1 of [MOV, MOVS \(register\)](#).

The value written to the PC is forced to be halfword-aligned by ignoring its least significant bit, treating that bit as being 0.
- The B, BL, CBNZ, CBZ, TBB, and TBH instructions remain in the same instruction set state and branch to the value written to the PC.

The definition of each of these instructions ensures that the value written to the PC is correctly aligned for the current instruction set state.
- The BLX (immediate) instruction switches between T32 and A32 states and branches to the value written to the PC. Its definition ensures that the value written to the PC is correctly aligned for the new instruction set state.
- The following instructions write a value to the PC, treating that value as an interworking address to branch to, with low-order bits that determine the new instruction set state:
 - BLX (register), BX, and BXJ.
 - LDR instructions with <Rt> equal to the PC.
 - POP and all forms of LDM except LDM (exception return), when the register list includes the PC.
 - In A32 state only, ADC, ADD, ADR, AND, ASR (immediate), BIC, EOR, LSL (immediate), LSR (immediate), MOV, MVN, ORR, ROR (immediate), RRX, RSB, RSC, SBC, and SUB instructions with <Rd> equal to the PC and without flag-setting specified.

For details of how an interworking address specifies the new instruction set state and instruction address, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

Note

The register-shifted register instructions, which are available only in the A32 instruction set and are summarized in [Data-processing register \(register shift\)](#), are CONSTRAINED UNPREDICTABLE if they attempt to write to the PC, see [Using R15 by instruction](#).

- Some instructions are treated as exception return instructions, and write both the PC and the [CPSR](#). For more information, including which instructions are exception return instructions, see [Exception return](#).
- Some instructions cause an exception, and the exception handler address is written to the PC as part of the exception entry.

B1.2.3.2 Pseudocode description of operations on the AArch32 general-purpose registers and the PC

In pseudocode, the uses of the [R\(\)](#) accessor, with an index argument *n*, are:

- Reading or writing R0-R12, SP, and LR, using *n* = 0-12, 13, and 14 respectively.
- Reading the PC, using *n* = 15.

[Pseudocode description of general-purpose register and PC operations](#) describes accesses to these registers.

Descriptions of A32 store instructions that store the PC value use the [PCStoreValue\(\)](#) pseudocode function to specify the PC value stored by the instruction.

Writing an address to the PC causes either a simple branch to that address or an *interworking* branch that also selects the instruction set to execute after the branch. A simple branch is performed by the [BranchWritePC\(\)](#) function.

An interworking branch is performed by the [BXWritePC\(\)](#) function.

The [LoadWritePC\(\)](#) and [ALUWritePC\(\)](#) functions are used for two cases where the behavior was systematically modified between architecture versions.

B1.2.4 Process state, PSTATE

Process state or [PSTATE](#) is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

Note

In this chapter, references to [PSTATE](#) link to the more appropriate of:

- The Application-level view of [PSTATE](#) given in this section.
 - The System-level description in [Process state, PSTATE](#).
-

The following [PSTATE](#) information is accessible at EL0:

The condition flags

Flag-setting instructions set these. They are:

N	Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to: • 1 if the result is negative. • 0 if the result is positive or zero.
Z	Zero Condition flag. Set to: • 1 if the result of the instruction is zero. • 0 otherwise. A result of zero often indicates an equal result from a comparison.
C	Carry Condition flag. Set to: • 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition. • 0 otherwise.
V	Overflow Condition flag. Set to: • 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition. • 0 otherwise.

Conditional instructions test the N, Z, C, and V Condition flags, combining them with the *Condition code* for the instruction, to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see [Conditional execution](#).

The overflow or saturation flag

Q	Some instructions can set this. For those instructions that can, the PE: • Sets it to 1 if the instruction indicates overflow or saturation. • Leaves it unchanged otherwise.
----------	--

For more information, see [Pseudocode description of saturation](#).

The greater than or equal flags

GE[3:0]	The instructions described in Parallel addition and subtraction instructions update these to indicate the results from individual bytes or halfwords of the operation. These flags can control a later <code>SEL</code> instruction. For more information, see SEL .
----------------	--

PSTATE also contains *PE state controls*. There is no direct access to these from application level instructions, but they can be changed by side-effects of application level instructions. They are:

Instruction set state

J, T The current instruction set state, as shown in [Table B1-1](#). The J bit is RES0, see the Note in this section.

Table B1-1 PSTATE.{J, T} encoding

J	T	Instruction set state
0	0	A32
0	1	T32

A32 The PE is executing the A32 instruction set, summarized in [A32 Instruction Set Encoding](#).

T32 The PE is executing the T32 instruction set, summarized in [T32 Instruction Set Encoding](#).

Note

Encoding with J==1 before Armv8, Jazelle, and T32EE states

In previous versions of the Arm architecture, the encoding {1, 0} selected Jazelle state, and encoding {1, 1} selected T32EE state. From the introduction of Armv8, the architecture does not support either of these states, and these are encodings for unimplemented instruction set states, see [Unimplemented instruction sets](#). AArch32 state requires a Trivial Jazelle implementation, see [Trivial implementation of the Jazelle extension](#).

The IT block state

IT[7:0] The If-Then controls for the T32 **IT** instruction, which applies to the *IT block* of instructions that immediately follow the **IT** instruction. See [IT](#) for a description of the **IT** instruction and its associated **IT** block.

For more information about the use of **PSTATE.IT**, see [Use of PSTATE.IT](#).

Endianness mapping

E For data accesses, controls the endianness:

0 Little-endian.

1 Big-endian.

If an implementation does not provide:

- Big-endian support for data accesses, this bit is RES0.
- Little-endian support for data accesses, this bit is RES1.

Instruction fetches are always little-endian, and ignore **PSTATE.E**.

B1.2.4.1 Accessing PSTATE fields at EL0

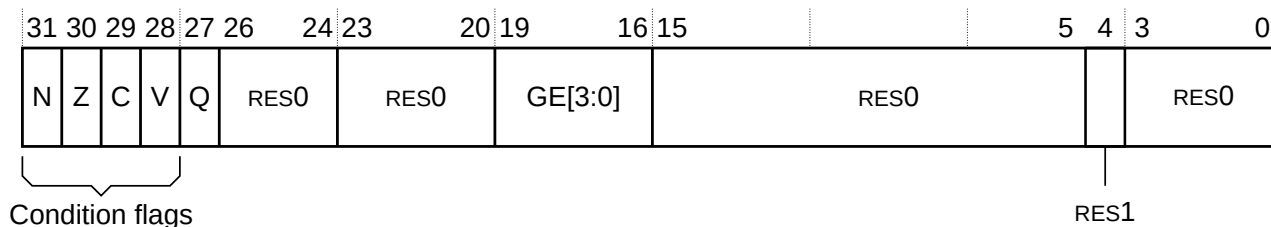
The following sections describe which **PSTATE** fields can be directly accessed at EL0, and how they can be accessed:

- [The Application Program Status Register, APSR](#).
- [The SETEND instruction](#).

B1.2.4.1.1 The Application Program Status Register, APSR

At EL0, some [PSTATE](#) fields can be accessed using the Special-purpose *Application Program Status Register* ([APSR](#)). The APSR can be directly read using the MRS instruction, and directly written using the [MSR \(register\)](#) and [MSR \(immediate\)](#) instructions.

The APSR bit assignments are:



N, Z, C, V, bits [31:28]

The [PSTATE](#) Condition flags.

Q, bit [27] The [PSTATE](#) overflow or saturation flag.

Bits[26:24] Reserved, RES0. Software can use MSR instructions that write the top byte of the [APSR](#) without using a read-modify-write sequence. If it does this, it must write zeros to bits[26:24].

Bits[23:20, 15:0]

Reserved bits that are allocated to system features, or are available for future expansion. Unprivileged execution ignores writes to fields that are accessible only at EL1 or higher. However, application level software that writes to the APSR must treat reserved bits as *Do-Not-Modify* (DNM) bits. For more information about the reserved bits, see [The Current Program Status Register, CPSR](#).

These bits are UNKNOWN on a Read, and it is permitted that, on a read of [APSR](#):

- Bit[22] returns the value of [PSTATE](#).PAN.
- Bit[9] returns the value of [PSTATE](#).E.
- Bits[8:6] return the value of [PSTATE](#).{A, I, F}, the mask bits.
- Bits[4:0] return the value of [PSTATE](#).M[4:0]. Bit[4] is RES1 indicating that the PE is in AArch32 state.

Note

This is an exception to the general rule that an UNKNOWN field must not return information that cannot be obtained, at the current Privilege level, by an architected mechanism.

GE[3:0], bits [19:16]

The [PSTATE](#) greater than or equal flags.

The other [PSTATE](#) fields cannot be accessed by using the APSR.

The system level alias for the APSR is the [CPSR](#). The [CPSR](#) is a superset of the APSR. See [The Current Program Status Register, CPSR](#).

Writes to the [PSTATE](#) fields have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the APSR write. This means they are guaranteed:

- Not to be visible to earlier instructions in the Execution stream.
- To be visible to later instructions in the Execution stream.

B1.2.4.1.2 The SETEND instruction

The T32 and A32 instruction sets both include an instruction to manipulate [PSTATE.E](#):

SETEND BE Sets [PSTATE.E](#) to 1, for big-endian operation.

SETEND LE Sets [PSTATE.E](#) to 0, for little-endian operation.

The SETEND instruction is unconditional. For more information, see [SETEND](#). Arm deprecates use of the SETEND instruction.

B1.2.4.2 Use of PSTATE.IT

[PSTATE.IT](#) provides the If-Then controls for the T32 IT instruction, which applies to the *IT block* of instructions that immediately follow the IT instruction.

[PSTATE.IT](#) divides into two subfields:

- IT[7:5]** Holds the *base condition* for the current IT block. The base condition is the top three bits of the Condition code specified by the <firstcond> field of the IT instruction.
- IT[4:0]** Encodes:
- Implicitly, the size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is indicated by the position of the least significant 1 in this field, as shown in [Table B1-2](#).
 - For each instruction in the IT block, the least significant bit of the Condition code. This is encoded in the IT block entries that [Table B1-2](#) shows as Nx.

Note

Changing the least significant bit of a Condition code from 0 to 1 has the effect of inverting the Condition code.

Both subfields are all zeros when no IT block is active.

When an IT instruction is executed, [PSTATE.IT](#) is set according to the <firstcond> field of the instruction and the *Then* and *Else* (T and E) parameters in the instruction, see [IT](#). This means that, on executing an IT instruction, the initial state of [PSTATE.IT](#) depends on the number of instructions in the IT block, as [Table B1-2](#) shows:

Table B1-2 Initial state of PSTATE.IT on executing an IT instruction

Number of instructions in IT block	PSTATE.IT bits ^a						Notes
	[7:5]	[4]	[3]	[2]	[1]	[0]	
4	cond_base	N1	N2	N3	N4	1	-
3	cond_base	N1	N2	N3	1	0	-
2	cond_base	N1	N2	1	0	0	-
1	cond_base	N1	1	0	0	0	-
Not executing an IT instruction	000	0	0	0	0	0	No IT block is active

a. Combinations of the IT bits not shown in this table are reserved.

In [Table B1-2](#), N1 refers to the first instruction in the IT block, and N2, N3, and N4 refer to the second, third, and fourth instructions in the IT block if they are present,

When permitted, an instruction in an IT block is conditional, see [Conditional instructions](#) and [Conditional execution](#). The Condition code used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, [PSTATE.IT](#)[4:0] is left-shifted by one bit, so that [PSTATE](#)[4] always relates to the next instruction to be executed.

Table B1-3 shows how `PSTATE.IT` during the execution of an `IT` instruction with four instructions in the IT block.

Table B1-3 Updates to `PSTATE.IT` when executing an IT instruction with a four-instruction IT block

IT block instruction being executed	PSTATE.IT bits ^a						Notes
	[7:5]	[4]	[3]	[2]	[1]	[0]	
First	cond_base	N1	N2	N3	N4	1	-
Second	cond_base	N2	N3	N4	1	0	-
Third	cond_base	N3	N4	1	0	0	-
Fourth	cond_base	N4	1	0	0	0	-
Not executing an <code>IT</code> instruction	000	0	0	0	0	0	No IT block is active

a. Combinations of the IT bits not shown in this table are reserved.

A few instructions, for example `BKPT`, cannot be conditional and therefore are always executed ignoring the current value of `PSTATE.IT`.

For details of what happens if an instruction in an IT block takes an exception, see [Overview of exception entry](#).

An instruction that might complete its normal execution by branching is only permitted in an IT block as the last instruction in the block. This means that normal execution of the instruction always results in `PSTATE.IT` advancing to execution where no IT block is active.

Implementations can provide a set of ITD control fields, `SCTLR.ITD`, and `HSCTLR.ITD`, to disable use of IT for some instructions, making them `UNDEFINED`. When an implementation includes ITD control fields, [Changes to an ITD control by an instruction in an IT block](#) describes the permitted `CONSTRAINED UNPREDICTABLE` behaviors if an instruction in an IT block changes the value of an ITD control to disable the use of the `IT` instruction.

On a branch or an exception return, if `PSTATE.IT` is set to a value that is not consistent with the instruction stream being branched to or returned to, then instruction execution is `CONSTRAINED UNPREDICTABLE`.

`PSTATE.IT` affects instruction execution only in T32 state. In A32 state, `PSTATE.IT` must be `0b00000000`, otherwise the behavior is `CONSTRAINED UNPREDICTABLE`.

For more information, see [CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT](#).

B1.2.4.2.1 Changes to an ITD control by an instruction in an IT block

In an implementation that includes `SCTLR.ITD`, and `HSCTLR.ITD` controls, if an instruction in an IT block changes an ITD control so that the `IT` instruction using the IT block would be disabled, then one of the following behaviors applies:

- The change to the ITD field, once synchronized, has no effect on the execution of instructions in the current IT block, but applies only to any subsequent execution of an `IT` instruction to which the control applies.
- Synchronizing the change to the ITD field guarantees that all bits of `PSTATE.IT` are cleared to 0.

In addition, after the change to the ITD field has been synchronized, any remaining instructions in the IT block that would be made `UNDEFINED` by the new value of ITD are either:

- Executed normally.
- Treated as `UNDEFINED`.

The choice between the options described in this section is determined by the implementation, and any choice can vary between different changes to an ITD control by an instruction in an IT block.

B1.2.4.3 Pseudocode description of PSTATE PE state fields

The pseudocode function `CurrentInstrSet()` returns the current instruction set. The pseudocode function `SelectInstrSet()` selects a new instruction set.

`PSTATE.IT` advances after normal execution of an IT block instruction. This is described by the `AArch32_ITAdvance()` pseudocode function.

The pseudocode function `InITBlock()` tests whether the current instruction is in an IT block. The pseudocode function `LastInITBlock()` tests whether the current instruction is the last instruction in an IT block.

The `BigEndian()` pseudocode function tests whether big-endian data memory accesses are currently selected.

B1.2.5 Jazelle support

The architecture requires AArch32 state to include a trivial implementation of the Jazelle Extension, as described in [Trivial implementation of the Jazelle extension](#).

B1.3 Advanced SIMD and floating-point instructions

The Armv8-R AArch32 architecture supports Advanced SIMD and floating-point operations. Armv8-R can optionally support single-precision floating-point operations without Advanced SIMD.

The Advanced SIMD instructions perform packed *Single Instruction Multiple Data* (SIMD) operations, on either integer or single-precision floating-point operands. The floating-point instructions perform single-precision or double-precision scalar floating-point operations.

These instructions permit *floating-point exceptions*, such as Overflow or Divide by Zero, to be handled without trapping. When handled in this way, a floating-point exception causes a cumulative status register bit to be set to 1 and a default result to be produced by the operation. The architecture also optionally supports the trapping of floating-point exceptions. For more information about floating-point exceptions, see [Floating-point exceptions and exception traps](#).

The Advanced SIMD and floating-point instructions also provide the following conversion functions:

- Between half-precision floating-point and single-precision floating-point, in both directions.
- From double-precision, floating-point to single-precision floating-point or integer.

Some Advanced SIMD instructions support polynomial arithmetic over $\{0, 1\}$, as described in Polynomial arithmetic over $\{0, 1\}$.

For system level information about the Advanced SIMD and floating-point implementation, see [Advanced SIMD and floating-point support](#).

The following sections give more information about the Advanced SIMD and floating-point instructions:

- [Floating-point standards, and terminology](#).
- [The SIMD and floating-point register file](#).
- [Data types supported by the Advanced SIMD implementation](#).
- [Advanced SIMD and floating-point System registers](#).
- [Floating-point data types and arithmetic](#).
- [Flushing denormalized numbers to zero](#).
- [Floating-point exceptions and exception traps](#).
- [Controls of Advanced SIMD operation that do not apply to floating-point operation](#).
- [Implications of not including Advanced SIMD and floating-point support](#).
- [Pseudocode description of floating-point operations](#).

B1.3.1 The SIMD and floating-point register file

The Advanced SIMD and floating-point instructions use the same register file, which comprises 32 registers. This is distinct from the register file that holds the general-purpose registers and the PC.

The Advanced SIMD and floating-point views of the register file are different. The following sections describe these different views. [Figure B1-1](#) shows the views of the register file, and the way the word, doubleword, and quadword registers overlap.

B1.3.1.1 Advanced SIMD views of the register file

Advanced SIMD can view this register file as:

- Sixteen 128-bit quadword registers, Q0–Q15.
- Thirty-two 64-bit doubleword registers, D0–D31.

These views can be used simultaneously. For example, a program might hold 64-bit vectors in D0 and D1 and a 128-bit vector in Q1.

B1.3.1.2 Floating-point views of the register file

The Advanced SIMD and floating-point register file consists of thirty-two doubleword registers, which can be viewed as:

- Thirty-two 64-bit doubleword registers, D0–D31. This view is also available to Advanced SIMD instructions.
- Thirty-two 32-bit single word registers, S0–S31. Only half of the set is accessible in this view.

Note
In AArch32 state, half-precision floating point values are always represented using the bottom 16 bits of a single word register, S0–S31. When a half-precision value is written to a single word register, the top 16 bits of that register are set to 0.

The two views can be used simultaneously.

B1.3.1.3 SIMD and Floating-point register file mapping onto registers

Figure B1-1 shows the different views of the SIMD and floating-point register file, and the relationship between them.

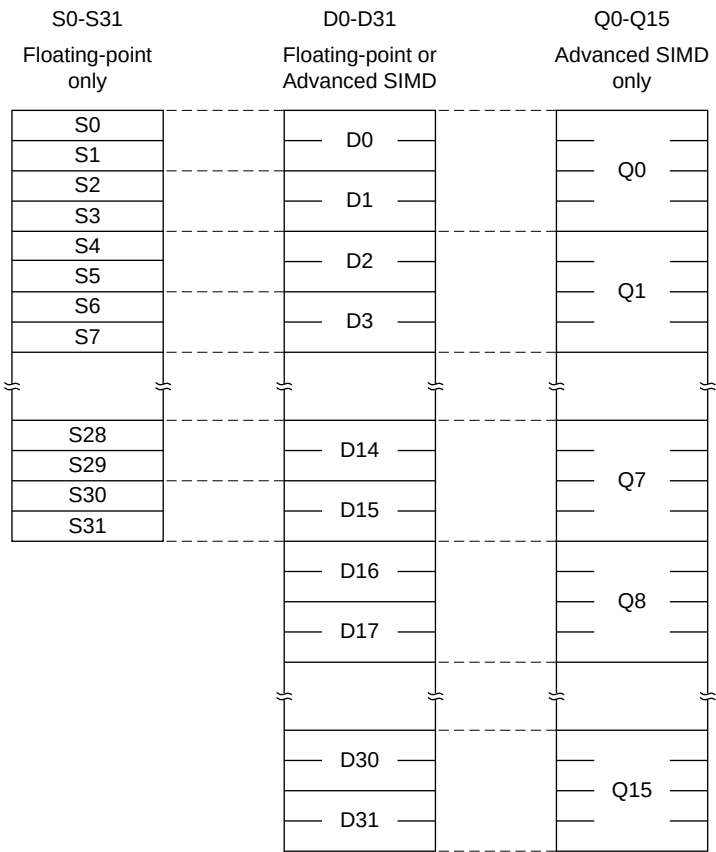


Figure B1-1 SIMD and floating-point register file, AArch32 operation

The mapping between the registers is as follows:

- S<2n> maps to the least significant half of D<n>.
- S<2n+1> maps to the most significant half of D<n>.
- D<2n> maps to the least significant half of Q<n>.
- D<2n+1> maps to the most significant half of Q<n>.

For example, software can access the least significant half of the elements of a vector in Q6 by referring to D12, and the most significant half of the elements by referring to D13.

B1.3.1.4 Pseudocode description of the SIMD and Floating-point register file

The array `_Dclone`, accessors `S()`, and `D()` provide the S0-S31, D0-D31, and Q0-Q15 views of the Advanced SIMD and floating-point registers. The array `_Dclone` is used to hold a copy of the Advanced SIMD and VFP registers.

The `Din()` function returns a doubleword register from the `_Dclone` copy of the SIMD and Floating-point register file, and the `Qin()` function returns a quadword register from that register file.

Note

The `CheckAdvSIMDEnabled()` function copies the `D()` register file to `_Dclone`, see [Pseudocode description of enabling SIMD and floating-point functionality](#).

B1.3.2 Data types supported by the Advanced SIMD implementation

Advanced SIMD instructions can operate on integer and floating-point data, and the implementation defines a set of data types that support the required data formats. [Advanced SIMD vector formats in AArch32 state](#) describes these formats.

B1.3.2.1 Advanced SIMD vectors

In an implementation that includes support for Advanced SIMD operation, a register can hold one or more packed elements, all of the same size and type. The combination of a register and a data type describes a vector of elements. The vector is considered to be an array of elements of the data type specified in the instruction. The number of elements in the vector is implied by the size of the data elements and the size of the register.

Vector indices are in the range 0 to (number of elements - 1). An index of 0 refers to the least significant end of the vector. In [Advanced SIMD vector formats in AArch32 state](#), [Figure A1-2](#) shows the Advanced SIMD vector formats.

B1.3.3 Advanced SIMD and floating-point System registers

The Advanced SIMD and floating-point instructions have a shared register space for System registers. The only register in this space that is accessible at the Application level is the [FPSCR](#).

Writes to the [FPSCR](#) can have side-effects on various aspects of PE operation. All of these side-effects are synchronous to the [FPSCR](#) write. This means they are guaranteed not to be visible to earlier instructions in the Execution stream, and they are guaranteed to be visible to later instructions in the Execution stream.

See [Advanced SIMD and floating-point System registers](#) for the system level view of the registers.

These registers can be described as the *SIMD and floating-point System registers*.

B1.3.4 Floating-point data types and arithmetic

The T32 and A32 floating-point instructions support single-precision (32-bit) and double-precision (64-bit) data types and arithmetic as defined by the IEEE 754 floating-point standard. They also support the half-precision (16-bit) floating-point data type for data storage, by supporting conversions between single-precision and half-precision data types.

Arm standard floating-point arithmetic means IEEE 754 floating-point arithmetic with the restrictions described in [Floating-point support](#), including supporting only the input and output values described in [Arm standard floating-point input and output values](#).

The following sections describe the Advanced SIMD and floating-point formats:

- [Half-precision floating-point formats](#).
- [Single-precision floating-point format](#).
- [Double-precision floating-point format](#).

The following sections describe features of Advanced SIMD and floating-point processing:

- [Flushing denormalized numbers to zero](#).
- [NaN handling and the default NaN](#).

B1.3.5 Flushing denormalized numbers to zero

Calculations involving denormalized numbers and Underflow exceptions can reduce the performance of floating-point processing. For many algorithms, replacing the denormalized operands and Intermediate results with zeros can recover this performance, without significantly affecting the accuracy of the final result. Arm floating-point implementations allow denormalized numbers to be flushed to zero to permit this optimization. If a number satisfies the condition $0 < \text{Abs}(\text{result}) < \text{MinNorm}$, it is treated as a denormalized number.

MinNorm is defined as follows:

- For half-precision numbers, MinNorm is 2^{-14} .
- For single-precision, MinNorm is 2^{-126} .
- For double-precision numbers, MinNorm is 2^{-1022} .

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

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

B1.3.5.1 Flushing denormalized inputs to zero

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

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

For Advanced SIMD and scalar floating-point instructions that process single-precision, or double-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If **FPSCR.FZ** is 0, flushing denormalized inputs to zero occurs as follows:
 - For Advanced SIMD floating-point instructions, all single-precision and double-precision inputs that are denormalized numbers are flushed to zero.
 - For scalar floating-point instructions, single-precision and double-precision inputs that are denormalized numbers are not flushed to zero.
- If **FPSCR.FZ** is 1, for all T32 and A32 instructions, single-precision and double-precision inputs that are denormalized numbers are flushed to zero.

B1.3.5.2 Flushing denormalized outputs to zero

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

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

For Advanced SIMD and scalar floating-point instructions that process single-precision, or double-precision inputs, flushing denormalized outputs to zero can be controlled as follows:

- If **FPSCR.FZ** is 0, flushing denormalized outputs to zero occurs as follows:
 - For Advanced SIMD floating-point instructions, all single-precision and double-precision outputs that are denormalized numbers are flushed to zero.
 - For scalar floating-point instructions, single-precision and double-precision outputs that are denormalized numbers are not flushed to zero.
- If **FPSCR.FZ** is 1, for all T32 and A32 instructions, single-precision and double-precision outputs that are denormalized numbers are flushed to zero.

B1.3.6 NaN handling and the default NaN

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

The Arm architecture additionally defines a *default NaN*, compliant with the IEEE 754 standard, that follows the encoding in [Table B1-4](#)

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

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

B1.3.6.1 The default NaN

The default NaN is encoded as described in [Table B1-4](#).

Table B1-4 Default NaN encoding

	Single-precision IEEE format	Double-precision
Sign bit	0	0
Exponent	0xFF	0x7FF
Fraction	Bit[22] == 1, bits[21:0] == 0	Bit[51] == 1, bits[50:0] == 0

If [FPSCR.DN](#) is 1, for Advanced SIMD and floating-point instructions if any input to a floating-point operation performed by the instruction is a NaN, the output of the floating-point operation is the default NaN.

If [FPSCR.DN](#) is 0, for floating-point processing the default NaN is not used for NaN propagation.

If a floating-point instruction performs a floating-point operation, and that instruction generates an untrapped Invalid Operation floating-point exception for a reason other than one of the inputs being a signaling NaN, the output is the default NaN.

B1.3.6.2 NaN handling

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

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

B1.3.6.3 NaN propagation

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

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

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN.
- If the input NaN is a signaling NaN, the output NaN is derived as follows:
 - If the handling of a signaling NaN by the instruction generates an Invalid Operation exception, the output NaN is the quieted version of the input NaN.
 - If the handling of a signaling NaN by the instruction does not generate an Invalid Operation exception, the output NaN is the same as the input NaN.

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

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

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

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

For the following descriptions, when an operand is described as *first* this relates to the left-to-right ordering of the arguments of the pseudocode function that describes the operation.

If `FPSCR.DN` is 0, for Advanced SIMD or floating-point instructions that perform a floating-point operation, NaN outputs that derive from NaN inputs are derived as follows:

- If all of the following apply, an instruction outputs a quiet NaN derived from the first signaling NaN operand:
 - At least one operand is a signaling NaN.
 - The instruction is not trapped.
- If all of the following apply, an instruction outputs a quiet NaN derived from the first NaN operand:
 - At least one operand is a NaN, but none of the operands is a signaling NaN.
 - The instruction is not trapped.

B1.3.7 Rounding

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

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

The rounding mode control field `FPSCR.RMode` can select the following rounding modes:

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

The following two additional rounding modes are not selected by `FPSCR.RMode`, but are used by some instructions:

- Round to Odd mode.
- Round to Nearest with ties to away mode.

B1.3.7.1 Round to Nearest mode

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

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

Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the `RMode` bits.

B1.3.7.2 Round towards Plus Infinity mode

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

B1.3.7.3 Round towards Minus Infinity mode

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

B1.3.7.4 Round towards Zero mode

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

B1.3.7.5 Round to Nearest with Ties to Away

Round to Nearest with Ties to Away rounding mode is used by the [VCVTA \(Advanced SIMD\)](#), [VCVTA \(floating-point\)](#), [VRINTA \(Advanced SIMD\)](#), and [VRINTA \(floating-point\)](#) instructions.

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

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

B1.3.7.6 Round to Odd mode

Round to Odd mode is not defined by IEEE 754.

B1.3.8 Floating-point exceptions and exception traps

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

Note

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

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

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

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

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

If an implementation supports synchronous exception generation for a floating-point exception, then the registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception.

On return from a synchronous floating-point exception, software might not restore the cumulative exception flags.

Trapped floating-point exceptions are taken to the following levels:

- If a trapped floating-point exception occurs at EL0, the exception is taken to EL1.
- If a trapped floating-point exception occurs at EL1, it is taken to EL1.
- If a trapped floating-point exception occurs at EL2, it is taken to EL2.

If the exception is taken, then it is taken as an Undefined Instruction exception, see [Undefined Instruction exception](#). The [FPEXC](#) identifies the floating-point exceptions that occurred since the corresponding status bits in that register were last set to 0.

B1.3.8.1 Input Denormal exceptions

The cumulative floating-point exception bit [FPSCR.IDC](#), and the trap enable bit [FPSCR.IDE](#) both relate to Input Denormal exceptions.

If a single-precision or double-precision floating-point input is flushed to zero, an Input Denormal exception is generated.

If a half-precision floating-point value is flushed to zero, an Input Denormal exception is not generated.

B1.3.8.2 Inexact exceptions

The cumulative floating-point exception bit [FPSCR.IXC](#), and the trap enable bit [FPSCR.IXE](#) both relate to Inexact exceptions.

If a denormalized output is flushed to zero, an Inexact exception is not generated.

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

B1.3.8.3 Underflow exceptions

The cumulative floating-point trap enable bit [FPSCR.UFE](#) relates to Underflow exceptions.

For the purpose of Underflow floating-point exception generation, a denormalized number is detected before rounding is applied.

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

- If the result is exact, an Underflow floating-point exception is generated only if [FPSCR.UFE](#) is 1.
- If the result is inexact, an Underflow floating-point exception is always generated.

If the result of a floating-point operation is a denormalized number that is flushed to zero, then the Underflow floating-point exception is generated. The Underflow exception is not trapped regardless of the value of [FPSCR.UFE](#).

B1.3.8.4 Overflow exceptions

The cumulative floating-point exception bit [FPSCR.OFC](#), and the trap enable bit [FPSCR.OFE](#) both relate to Overflow exceptions.

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

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

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

B1.3.8.5 Divide by Zero exceptions

The cumulative floating-point exception bit [FPSCR.DZC](#), and the trap enable bit [FPSCR.DZE](#) both relate to Divide by Zero exceptions.

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

For the purpose of Divide by Zero exception generation, testing for zero occurs after flushing of denormalized numbers to zero.

A denormalized dividend that is flushed to zero is treated as zero and prevents Divide by Zero from occurring.

If the dividend is a finite nonzero, normalized number, and the divisor is a denormalized number, the divisor is treated as zero and causes Divide by Zero to occur.

For the reciprocal and reciprocal square root estimate functions, the dividend is assumed to be +1.0. This means that a zero or denormalized operand to these functions causes generation of a Divide by Zero floating-point exception.

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

B1.3.8.6 Invalid Operation exceptions

The cumulative floating-point exception bit [FPSCR.IOC](#), and the trap enable bit [FPSCR.IOE](#) both relate to Invalid Operation exceptions.

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

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

For the purpose of Invalid Operation Exception generation, testing for zero occurs after flushing of denormalized numbers to zero. So a denormalized input that is flushed to zero is treated as zero.

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

B1.3.8.7 Floating-point exception traps

For Advanced SIMD instructions, and for floating-point instructions when floating-point exception trapping is not supported, these are non-trapping exceptions and the data-processing instructions do not generate any trapped exceptions.

For floating-point instructions when floating-point exception trapping is supported:

- The floating-point exceptions can be trapped, by setting trap enable bits in the [FPSCR](#), see [Floating-point exceptions and exception traps](#), and:
 - When a trap is not enabled the corresponding floating-point exception updates the corresponding [FPSCR](#) cumulative bit, but does not generate an exception.

- When a trap is enabled the corresponding floating-point exception does not update the [FPSCR](#), but generates an exception. In this case, bits in the [FPEXC](#) indicate which floating-point exceptions have occurred.
- The definition of the Underflow floating-point exception is different in the trapped and cumulative exception cases. In the trapped case, the definition is:
 - The trapped Underflow floating-point exception occurs if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, regardless of whether the rounded result is inexact.
- As with cumulative exceptions, higher priority trapped exceptions can prevent lower priority exceptions from occurring, as described in [Combinations of floating-point exceptions](#).
- For Invalid Operation floating-point exceptions, for details of which quiet NaN is produced as the default result, see [NaN handling and the default NaN](#).
- For Overflow floating-point exceptions, the sign bit of the default result is determined normally for the overflowing operation.
- For Divide by Zero floating-point exceptions, the sign bit of the default result is determined normally for a division. This means it is the exclusive-OR of the sign bits of the two operands.

[Table B1-5](#) shows the results of untrapped floating-point exceptions. That table uses the following abbreviations:

MaxNorm	The maximum normalized number of the destination precision.
RM	Round towards Minus Infinity mode, as defined in the IEEE 754 standard.
RN	Round to Nearest mode, as defined in the IEEE 754 standard.
RP	Round towards Plus Infinity mode, as defined in the IEEE 754 standard.
RZ	Round towards Zero mode, as defined in the IEEE 754 standard.

For more information about the IEEE 754 descriptions of the rounding modes, see [Floating-point standards, and terminology](#).

Table B1-5 Results of untrapped floating-point exceptions

Exception type	Default result for positive sign	Default result for negative sign
IOC, Invalid Operation	Quiet NaN	Quiet NaN
DZC, Divide by Zero	+infinity	-infinity
OFC, Overflow	RN, RP: RM, RZ: +infinity +MaxNorm	RN, RM: RP, RZ: -infinity -MaxNorm
UFC, Underflow	Normal rounded result	Normal rounded result
IXC, Inexact	Normal rounded result	Normal rounded result
IDC, Input Denormal	Normal rounded result	Normal rounded result

B1.3.8.8 Combinations of floating-point exceptions

Many pseudocode functions perform *floating-point operations*, including [FixedToFP\(\)](#), [FPAdd\(\)](#), [FPCompare\(\)](#), [FPCompareEQ\(\)](#), [FPCompareGE\(\)](#), [FPCompareGT\(\)](#), [FPDiv\(\)](#), [FPMax\(\)](#), [FPMin\(\)](#), [FPMul\(\)](#), [FPMulAdd\(\)](#), [FPRecipEstimate\(\)](#), [FPRecipStep\(\)](#), [FPRSqrtEstimate\(\)](#), [FPRSqrtStep\(\)](#), [FPSqrt\(\)](#), [FPSub\(\)](#), and [FPToFixed\(\)](#). All of these operations can generate floating-point exceptions.

Note

[FPAbs\(\)](#) and [FPNeg\(\)](#) are not classified as *floating-point operations* because:

- They cannot generate floating-point exceptions.
 - The floating-point operation behavior described in the following sections does not apply to them:
 - [Flushing denormalized numbers to zero.](#)
 - [NaN handling and the default NaN.](#)
-

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

- Overflow with Inexact.
- Underflow with Inexact.
- Input Denormal with one or more of the other floating-point exception types.

The priority order of these floating-point exceptions is that the Inexact exception is treated as lowest priority, and the Input Denormal exception is treated as highest priority.

When none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative bit in the [FPSCR](#) to be set.

When one or more floating-point exceptions caused by an operation is trapped, the behavior of the instruction depends on the priority of the exceptions:

- If the higher priority floating-point exception is trapped, its trap handler is called. It is IMPLEMENTATION DEFINED whether any information about the lower priority floating-point exception is provided.

Note

Information about the lower priority floating-point exception might be provided in the [FPEXC](#).

However, information might be provided in another IMPLEMENTATION DEFINED way, for example using an IMPLEMENTATION DEFINED register.

Apart from this, the lower priority floating-point exception is ignored in this case.

- If the higher priority floating-point exception is untrapped, its cumulative bit is set to 1 and its default result is evaluated. Then the lower priority floating-point exception is handled normally, using this default result.

Some floating-point instructions specify more than one floating-point operation, as indicated by the pseudocode descriptions of the instruction. In such cases, a floating-point exception on one operation is treated as higher priority than a floating-point exception on another operation if the occurrence of the second floating-point exception depends on the result of the first operation. Otherwise, it is CONstrained UNPREDICTABLE which floating-point exception is treated as higher priority.

For example, a `VMLA.F32` instruction specifies a floating-point multiplication followed by a floating-point addition. The addition can generate Overflow, Underflow and Inexact floating-point exceptions, all of which depend on both operands to the addition and so are treated as lower priority than any floating-point exception on the multiplication. The same applies to Invalid Operation floating-point exceptions on the addition caused by adding opposite-signed infinities. The addition can also generate an Input Denormal floating-point exception, caused by the addend being a denormalized number while in Flush-to-zero mode. It is CONstrained UNPREDICTABLE which of an Input Denormal floating-point exception on the addition and a floating-point exception on the multiplication is treated as higher priority, because the occurrence of the Input Denormal floating-point exception does not depend on the result of the multiplication. The same applies to an Invalid Operation floating-point exception on the addition caused by the addend being a signaling NaN.

Note

The `VFMA` instruction performs a vector addition and a vector multiplication as a single operation. The `VFMS` instruction performs a vector subtraction and a vector multiplication as a single operation.

B1.3.9 Controls of Advanced SIMD operation that do not apply to floating-point operation

Earlier architectures permitted implementation of either, both, or neither of the Advanced SIMD and floating-point additions to the base instruction set, and provided some controls that applied to the Advanced SIMD functionality but not to the floating-point functionality. From the introduction of Armv8, Advanced SIMD functionality cannot be separated from floating-point functionality, but in AArch32 state these controls function as they did in earlier architectures. This means they apply only to the following instructions and instruction encodings:

- All instructions with encodings defined in:
 - [Advanced SIMD data-processing](#), for the T32 instruction set.
 - [Advanced SIMD data-processing](#), for the A32 instruction set.
- All instructions with encodings defined in:
 - [Advanced SIMD element or structure load/store](#), for the T32 instruction set.
 - [Advanced SIMD element or structure load/store](#), for the A32 instruction set.
- The form of the `VDUP` instruction described in [VDUP \(general-purpose register\)](#).
- The byte and halfword forms of the `VMOV` instructions described in each of:
 - [VMOV \(general-purpose register to scalar\)](#).
 - [VMOV \(scalar to general-purpose register\)](#).

The optional control for this functionality is the `HCPTR.TASE` field.

If the `HCPTR.TASE` control is not supported then the `HCPTR.TASE` field is RAZ/WI. This means the `HCPTR` does not provide a control that can trap execution of Advanced SIMD instructions to Hyp mode.

B1.3.10 Implications of not including Advanced SIMD and floating-point support

In general, the architecture requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. Exceptionally, for implementation targeting specialized markets, Arm might produce or license an implementation that does not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- Each of the `CPACR.{cp10, cp11}` fields is RES0.
- Each of the `HCPTR.{TASE, TCP10, TCP11}` fields is RES1.
- The `FPEXC` register is UNDEFINED.

B1.3.11 Pseudocode description of floating-point operations

The following subsections contain pseudocode definitions of the floating-point functionality supported by the architecture:

- [Generation of specific floating-point values.](#)
- [Floating-point negation and absolute value.](#)
- [Floating-point value unpacking.](#)
- [Floating-point exception and NaN handling.](#)
- [Floating-point rounding.](#)
- [Selection of Arm standard floating-point arithmetic.](#)
- [Floating-point comparisons.](#)
- [Floating-point maximum and minimum.](#)
- [Floating-point addition and subtraction.](#)
- [Floating-point multiplication and division.](#)
- [Floating-point fused multiply-add.](#)
- [Floating-point reciprocal estimate and step.](#)
- [Floating-point square root.](#)
- [Floating-point reciprocal square root estimate and step.](#)
- [Floating-point conversions.](#)

B1.3.11.1 Generation of specific floating-point values

The following pseudocode functions generate specific floating-point values. The `sign` argument is '0' for the positive version and '1' for the negative version:

- `FPInfinity()`.
- `FPMaxNormal()`.
- `FPZero()`.
- `FPTwo()`.
- `FPThree()`.
- `FPDefaultNaN()`.

B1.3.11.2 Floating-point negation and absolute value

The floating-point negation and absolute value operations affect only the sign bit. They do not treat NaN operands specially, nor denormalized number operands when flush-to-zero is selected.

The floating-point negation operation is described by the pseudocode function `FPNeg()`. The floating-point absolute value operation is described by the pseudocode function `FPAbs()`.

B1.3.11.3 Floating-point value unpacking

The `FPUnpack()` function determines the type of a floating-point number, defined by `FPType()`, and its numerical value. It also does flush-to-zero processing on input operands.

B1.3.11.4 Floating-point exception and NaN handling

The `FPProcessException()` procedure checks whether a floating-point exception is trapped, and handles it accordingly. The floating-point exception types are defined by `FPExc()`.

The `FPProcessNaN()` function processes a NaN operand, producing the correct result value and generating an Invalid Operation floating-point exception if necessary. The `FPProcessNaNs()` function performs the standard NaN processing for a two-operand operation. The `FPProcessNaNs3()` function performs the standard NaN processing for a three-operand operation.

B1.3.11.5 Floating-point rounding

The `FPRound()` function rounds and encodes a floating-point result to a specified destination format. This includes processing Overflow, Underflow and Inexact floating-point exceptions and performing flush-to-zero processing on result values.

B1.3.11.6 Selection of Arm standard floating-point arithmetic

The `StandardFPCR()` function returns the `FPSCR` value that selects Arm standard floating-point arithmetic. Most of the arithmetic functions have a Boolean `fpscr_controlled` argument that is `TRUE` for Floating-point operations and `FALSE` for Advanced SIMD operations, and that selects between using the real `FPSCR` value and this value.

B1.3.11.7 Floating-point comparisons

The `FPCompare()` function compares two floating-point numbers, producing a {N, Z, C, V} Condition flags result as shown in Table B1-6:

Table B1-6 Effect of a Floating-point comparison on the Condition flags

Comparison result	N	Z	C	V
Equal	0	1	1	0
Less than	1	0	0	0
Greater than	0	0	1	0

Comparison result	N	Z	C	V
Unordered	0	0	1	1

This result defines the operation of the `VCMP` floating-point instruction. The `VCMP` instruction writes these flag values in the `FPSCR`. After using a `VMRS` instruction to transfer them to the `APSR`, they can control conditional execution as shown in [Table C1-1](#).

The `FPCompareEQ()`, `FPCompareGE()`, and `FPCompareGT()` functions describe the operation of Advanced SIMD instructions that perform floating-point comparisons.

B1.3.11.8 Floating-point maximum and minimum

The `FPMAX()` function returns the maximum of two floating-point numbers. The `FPMIN()` function returns the minimum of two floating-point numbers.

B1.3.11.9 Floating-point addition and subtraction

The `FPMUL()` function adds two floating-point numbers. The `FPSUB()` function subtracts one floating-point number from another floating-point number.

B1.3.11.10 Floating-point multiplication and division

The `FPMUL()` function multiplies two floating-point numbers. The `FPDIV()` function divides one floating-point number by another floating-point number.

B1.3.11.11 Floating-point fused multiply-add

The `FPMULADD()` function performs a floating-point fused multiply-add.

B1.3.11.12 Floating-point reciprocal estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of a number.

The `VRECPE` instruction produces the initial estimate of the reciprocal. It uses the pseudocode functions:

- `FPRecipEstimate()`.
- `UnsignedRecipEstimate()`.

[Table B1-7](#) shows the results where input values are out of range.

Table B1-7 VRECPE results for out of range inputs

Number type	Input <code>Vm[i]</code>	Result <code>Vd[i]</code>
Integer	$\leq 0x7FFF_FFFF$	<code>0xFFFF_FFFF</code>
Floating-point	NaN	Default NaN
Floating-point	± 0 or denormalized number	$\pm\text{infinity}^a$
Floating-point	$\pm\text{infinity}$	± 0
Floating-point	Absolute value $\geq 2^{126}$	± 0

a. `FPSCR.DZC` is set to 1

The Newton-Raphson iteration:

$$x_{n+1} = x_n(2 - dx_n)$$

converges to $(1/d)$ if x_0 is the result of `VRECPE` applied to d .

The `VRECPS` instruction performs a $(2 - \text{op1} \times \text{op2})$ calculation and can be used with a multiplication to perform a step of this iteration. The functionality of this instruction is defined by the `FPrecipStep()` pseudocode function.

Table B1-8 shows the results where input values are out of range.

Table B1-8 VRECPS results for out of range inputs

Input Vn[i]	Input Vm[i]	Result Vd[i]
Any NaN	-	Default NaN
-	Any NaN	Default NaN
± 0.0 or denormalized number	$\pm$ infinity	2.0
$\pm$ infinity	± 0.0 or denormalized number	2.0

B1.3.11.13 Floating-point square root

The `FPSqrt()` function returns the square root of a floating-point number.

B1.3.11.14 Floating-point reciprocal square root estimate and step

The Advanced SIMD implementation includes instructions that support Newton-Raphson calculation of the reciprocal of the square root of a number.

The `VRSQRTE` instruction produces the initial estimate of the reciprocal of the square root. It uses the pseudocode functions:

- `FPRSqrtEstimate()`.
- `UnsignedRSqrtEstimate()`.

Table B1-9 shows the results where input values are out of range.

Table B1-9 VRSQRTE results for out of range inputs

Number type	Input Vm[i]	Result Vd[i]
Integer	$\leq 0 \times 3FFF_FFFF$	$0 \times FFFF_FFFF$
Floating-point	NaN, -(normalized number), -infinity	Default NaN
Floating-point	-0 or -(denormalized number)	- infinity ^a
Floating-point	+0 or +(denormalized number)	+infinity ^a
Floating-point	+infinity	+0

a. `FPSCR.DZC` is set to 1.

The Newton-Raphson iteration:

$$x_{n+1} = x_n(3 - dx_n^2)/2$$

converges to $(1/\sqrt{d})$ if x_0 is the result of `VRSQRTE` applied to d .

The `VRSQRTS` instruction performs a $(3 - \text{op1} \times \text{op2})/2$ calculation and can be used with two multiplications to perform a step of this iteration. The functionality of this instruction is defined by the `FPRSqrtStep()` pseudocode function.

Table B1-10 shows the results where input values are out of range.

Table B1-10 VRSQRTS results for out of range inputs

Input Vn[i]	Input Vm[i]	Result Vd[i]
Any NaN	-	Default NaN
-	Any NaN	Default NaN
± 0.0 or denormalized number	$\pm \text{infinity}$	1.5
$\pm \text{infinity}$	± 0.0 or denormalized number	1.5

`FPRSqrtStep()` calls the `FPHalvedSub()` pseudocode function.

B1.3.11.15 Floating-point conversions

The `FPConvert()` pseudocode function performs conversions between half-precision, single-precision, and double-precision floating-point numbers.

The `FPToFixed()` and `FixedToFP()` functions perform conversions between floating-point numbers and integers or fixed-point numbers.

B1.4 About the AArch32 System register interface

AArch32 state provides a System register encoding space, which is indexed by the parameter set {coproc, opc1, CRn, CRm, opc2}, and a set of System register access instructions. This encoding space is used for:

- System registers.
- System instructions, for:
 - Cache and branch predictor maintenance.
 - Address translation.

This encoding space uses only the coproc values 0b111x.

Note

The encoding space with coproc values 0b101x is redefined to provide Advanced SIMD and floating-point functionality.

The coproc encodings provide access to System register encoding space as follows:

- The (coproc==0b1111) encodings provide system control functionality, by providing access to System registers and System instructions. This includes architecture and feature identification, as well as control, status information and configuration support.
The following section gives a general description of these encodings: [Organization of registers in the \(coproc == 0b1111 \) encoding space](#).
These encodings also provide:
 - The Performance Monitor System registers. For more information, see [The Performance Monitors Extension](#).
- The (coproc==0b1110) encodings provide access to additional registers, which support:
 - Debug, see [AArch32 Self-hosted Debug](#).
 - The Jazelle identification registers, see [Jazelle support](#).

[UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses](#) gives more information about permitted accesses to the System registers in AArch32 state.

Most functionality in the (coproc==0b111x) encoding space cannot be accessed by software executing at EL0. This manual clearly identifies those functions that can be accessed at EL0.

For more information:

- About this encoding space, including the naming of the parameters that index the space, see [The AArch32 System register interface](#).
- About the System interface access instructions, see [System register access instructions](#).

B1.5 Exceptions

The Arm architecture uses the following terms to describe various types of exceptional condition:

Exceptions In the Arm architecture, an *exception* causes entry to EL1 or EL2. On entry to the Exception level that the exception is taken to, it also causes entry to the PE mode in which the exception must be taken. Exception levels in the Armv8-R AArch32 architecture are based on PE modes. A software handler for the exception is then executed.

Note

The term *floating-point exception* does not use this meaning of *exception*. This term is described later in this list.

Exceptions include:

- Reset.
- Interrupts.
- Memory system aborts.
- Undefined instructions.
- Supervisor calls (SVCs), and Hypervisor calls (HVCs).
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in [Handling exceptions that are taken to an Exception level using AArch32](#). In an implementation that includes all the Exception levels, aspects that are visible to application level software are:

- The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a call to the operating system, or other system component that is accessible only at EL1.
- The `SMC` instruction is UNDEFINED.
- The `HVC` instruction causes a Hypervisor Call exception, but only if software execution is at EL1 or higher. Unprivileged software can only cause a Hypervisor Call exception by methods defined by the hypervisor, or by another component of the software system that executes at EL1 or higher.
- The `BKPT` instruction causes a Breakpoint Instruction exception, which is taken as a Prefetch Abort exception. This provides a mechanism for a debugger to insert breakpoints into unprivileged software, or for unprivileged software to make a call into a debugger that is accessible at EL1.
- The `WFI` (Wait for Interrupt) instruction provides a hint that nothing needs to be done until an interrupt or another WFI wakeup event occurs, see [Wait For Interrupt](#). This means the hardware might enter a low-power state until the wakeup event occurs.
- The `WFE` (Wait for Event) instruction provides a hint that nothing needs to be done until either an `SEV` instruction generates an event, or another WFE wakeup event occurs, see [Wait For Event and Send Event](#). This means the hardware might enter a low-power state until the wakeup event occurs.

Floating-point exceptions

These relate to exceptional conditions encountered during floating-point arithmetic, such as Divide by Zero or Overflow. For more information, see:

- [Floating-point exceptions and exception traps](#).
- The `FPEXC` and `FPSCR` register descriptions.
- ANSI/IEEE Std. 754, *IEEE Standard for Binary Floating-Point Arithmetic*.

Chapter B2

The AArch32 Application Level Memory Model

This chapter gives an application level description of the memory model for software executing in AArch32 state. This means it describes the memory model for execution in EL0 in the following sections:

- [About the Arm memory model](#)
- [Atomicity in the Arm architecture](#)
- [Definition of the memory model](#)
- [Ordering requirements defined by the formal concurrency model](#)
- [Additional ordering requirements outside of the scope of the formal concurrency model](#)
- [Endian support](#)
- [Caches and memory hierarchy](#)
- [Alignment support](#)
- [Memory types and attributes](#)
- [Mismatched memory attributes](#)
- [Synchronization and semaphores](#)

Note

In this chapter, System register names usually link to the description of the register in [AArch32 System Register Descriptions](#), for example [SCTLR](#).

Note

In the [Definition of the memory model](#) there is mention of Features and states that are not achievable in the Armv8-R AArch32 architecture in order to maintain concurrence with [formal concurrency model](#). The Memory Model should be interpreted in accordance with the Features and limitations of the architecture described in [Armv8-R AArch32 architecture extensions](#).

B2.1 About the Arm memory model

The Arm architecture is a weakly ordered memory architecture that permits the observation and completion of memory accesses in a different order from the program order. The following sections of this chapter provide the complete definition of the memory model, this introduction is not intended to contradict the definition found in those sections. In general, the basic principles of the memory model are:

- To provide a memory model that has similar weaknesses to those found in the memory models used by high-level programming languages such as C or Java. For example, by permitting independent memory accesses to be reordered as seen by other observers.
- To avoid the requirement for multi-copy atomicity in the majority of memory types.
- The provision of instructions and memory barriers to compensate for the lack of multi-copy atomicity in the cases where it would be needed.
- The use of address, data, and control dependencies in the creation of order so as to avoid having excessive numbers of barriers or other explicit instructions in common situations where some order is required by the programmer or the compiler.

This section contains:

- [Address space](#).
- [Memory type overview](#).

B2.1.1 Address space

Address calculations are performed using 32-bit registers. Supervisory software determines the valid address range.

Attempting to access an address that is not valid generates an [MPU fault](#).

Address calculations are performed modulo 2^{32} .

The result of an address calculation is UNKNOWN if it overflows or underflows the 32-bit address range A[31:0].

Memory accesses use the [MemA{size}\(\)](#), [MemO{size}\(\)](#), [MemU{size}\(\)](#), and [MemU_unpriv{size}\(\)](#) pseudocode accessors:

- The [MemA{size}\(\)](#) accessor makes an aligned access of the required type.
- The [MemO{size}\(\)](#) accessor makes an ordered access of the required type.
- The [MemU{size}\(\)](#) accessor makes an unaligned access of the required type
- The [MemU_unpriv{size}\(\)](#) accessor makes an unaligned, unprivileged access of the required type.

Each of these accessors calls the [Mem_with_type{size}\(\)](#) accessor, which specifies the required access. This calls [AArch32_MemSingle{size}\(\)](#), which performs an atomic, little-endian read.

The [AccessDescriptor](#) type defines the different access types and attributes.

Note

- The [AArch32 System Level Memory Model](#) includes descriptions of memory system features that are transparent to the application, including memory access, address translation, memory maintenance instructions, and alignment checking and the associated fault handling. These chapters also reference pseudocode descriptions of these operations.
 - For references to the pseudocode that relates to memory accesses, see [Basic memory access](#), [Unaligned memory access](#), and [Aligned memory access](#).
-

B2.1.2 Memory type overview

The architecture provides the following mutually-exclusive memory types:

Normal	This is generally used for the majority of memory operations, both read/write and read-only operations.
---------------	---

- Device** The Arm architecture forbids speculative reads of any type of Device memory. This means Device memory types are suitable attributes for read-sensitive locations.
- Locations of the memory map that are assigned to peripherals are usually assigned the Device memory attribute.
- Device memory has additional attributes that have the following effects:
- They prevent aggregation of reads and writes, maintaining the number and size of the specified memory accesses. See [Gathering](#).
 - They preserve the access order and synchronization requirements, both for accesses to a single peripheral and where there is a synchronization requirement on the observability of one or more memory write and read accesses. See [Reordering](#)
 - They indicate whether a write can be acknowledged other than at the end point. See [Early Write Acknowledgement](#).

For more information on Normal memory and Device memory, see [Memory types and attributes](#).

Note

Earlier versions of the Arm architecture defined a single Device memory type and a Strongly-ordered memory type. A Note in [Device memory](#) describes how these memory types map onto the memory types used from the introduction of Armv8.

B2.2 Atomicity in the Arm architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The Arm architecture description refers to two types of atomicity, *single-copy atomicity* and *multi-copy atomicity*. The atomicity requirements for memory accesses depend on the memory type, and whether the access is explicit or implicit. For more information, see:

- [Requirements for single-copy atomicity.](#)
- [Properties of single-copy atomic accesses.](#)
- [Multi-copy atomicity.](#)
- [Requirements for multi-copy atomicity.](#)
- [Concurrent modification and execution of instructions.](#)

For more information about the memory types, see [Memory type overview](#).

B2.2.1 Requirements for single-copy atomicity

In AArch32 state, the single-copy atomic PE accesses are:

- All byte accesses.
- All halfword accesses to halfword-aligned locations.
- All word accesses to word-aligned locations.
- Memory accesses caused by `LDREXD` and `STREXD` instructions to doubleword-aligned locations.

`LDM`, `LDC`, `LDRD`, `STM`, `STC`, `STRD`, `PUSH`, `POP`, `RFE`, `SRS`, `VLDM`, `VLDR`, `VSTM`, and `VSTR` instructions are executed as a sequence of word-aligned word accesses. Each 32-bit word access is guaranteed to be single-copy atomic. The architecture does not require subsequences of two or more word accesses from the sequence to be single-copy atomic.

`LDRD` and `STRD` accesses to 64-bit aligned locations are 64-bit single-copy atomic as seen by accesses.

Note

This requirement has been added to avoid the need for complex measures to avoid atomicity issues when changing memory protection regions, without creating a requirement that all locations in the memory system are 64-bit single-copy atomic. This addition means:

- The system designer must ensure that all writable memory locations that might be used to hold translations, such as bulk SDRAM, can be accessed with 64-bit single-copy atomicity.
- Software must ensure that memory protection region attributes are not held in memory locations that cannot meet this atomicity requirement, such as peripherals that are typically accessed using a narrow bus.

This requirement places no burden on read-only memory locations for which reads have no side effects, since it is impossible to detect the size of memory accesses to such locations.

Advanced SIMD element and structure loads and stores are executed as a sequence of accesses of the element or structure size. The architecture requires the element accesses to be single-copy atomic if and only if both:

- The element size is 32 bits, or smaller.
- The elements are naturally-aligned.

Accesses to 64-bit elements or structures that are 32-bit aligned are executed as a sequence of 32-bit accesses, each of which is single-copy atomic. The architecture does not require subsequences of two or more 32-bit accesses from the sequence to be single-copy atomic.

When an access is not single-copy atomic by the rules described in this section, it is executed as a sequence of one or more accesses that aggregate to the size of the original access. Each of the accesses in this sequence is single-copy atomic, at least at the byte level.

Note

In this section, the terms *before the write operation* and *after the write operation* mean before or after the write operation has had its effect on the coherence order of the bytes of the memory location accessed by the write operation.

If, according to these rules, an instruction is executed as a sequence of accesses, a synchronous Data Abort exception or Debug state entry can be taken during that sequence. This causes execution of the instruction to be abandoned.

See [Data Abort exception](#)

If the synchronous Data Abort exception is returned from using the preferred return address, the instruction that generated the sequence of accesses is re-executed and so any access that was performed before the exception was taken is repeated. This also applies to an exit from Debug state.

Note

The exception behavior for these multiple access instructions means they are not suitable for use for writes to memory for the purpose of software synchronization.

For implicit accesses:

- Cache linefills and evictions have no effect on the single-copy atomicity of explicit transactions or instruction fetches.
- Instruction fetches are single-copy atomic:
 - At 32-bit granularity in A32 state.
 - At 16-bit granularity in T32 state.[Concurrent modification and execution of instructions](#) describes additional constraints on the behavior of instruction fetches.
- Translations which are performed using accesses that are single-copy atomic at 64-bit granularity when using the Long-descriptor format.

B2.2.2 Properties of single-copy atomic accesses

A memory access instruction that is single-copy atomic has the following properties:

1. For a pair of overlapping single-copy atomic store instructions, all of the overlapping writes generated by one of the stores are Coherence-after the corresponding overlapping writes generated by the other store.
2. For a single-copy atomic load instruction L_1 that overlaps a single-copy atomic store instruction S_2 , if one of the overlapping reads generated by L_1 Reads-from-memory one of the overlapping writes generated by S_2 , then none of the overlapping writes generated by S_2 are Coherence-after the corresponding overlapping reads generated by L_1 .

For more information, see [Definition of the memory model](#).

B2.2.3 Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Note

Writes that are not coherent are not multi-copy atomic.

B2.2.4 Requirements for multi-copy atomicity

For Normal memory, writes are not required to be multi-copy atomic.

For Device memory, writes are not required to be multi-copy atomic.

The memory model is Other-multi-copy-atomic. For more information, see [External visibility requirement](#).

B2.2.5 Concurrent modification and execution of instructions

The architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where the instruction before modification and the instruction after modification are:

- When executing the A32 instruction set, a B, BKPT, BL, HVC, ISB, NOP, SMC, or SVC instruction.
- When executing the T32 instruction set, a 16-bit B, BKPT, BLX, BX, NOP, or SVC instruction.

In addition, for the 32-bit T32 instructions, for which [Instruction encodings](#) describes the meaning of {hw1, hw2}:

- hw1 of a 32-bit BL (immediate) instruction can be concurrently modified to hw1 of another BL (immediate) instruction:
 - This means that some of the most significant bits of the immediate value can be modified.
- hw1 of a 32-bit BLX (immediate) instruction can be concurrently modified to hw1 of another BLX immediate instruction:
 - This means that some of the most significant bits of the immediate value can be modified.
- hw1 of a 32-bit BL (immediate) or BLX (immediate) instruction can be concurrently modified to a T32 16-bit B, BX, BLX, BKPT, or SVC instruction. This modification also works in reverse.
- hw2 of a 32-bit BL (immediate) instruction can be concurrently modified to hw2 of another BL (immediate) instruction with a different immediate:
 - This means that some bits of the immediate value, including the least significant bits, can be modified.
- hw2 of a 32-bit BLX (immediate) instruction can be concurrently modified to hw2 of another BLX (immediate) instruction with a different immediate:
 - This means that some bits of the immediate value, including the least significant bits, can be modified.
- hw2 of a 32-bit B (immediate) instruction with a condition field can be concurrently modified to hw2 of another 32-bit B (immediate) instruction with a condition field with a different immediate:
 - This means that some bits of the immediate value, including the least significant bits, can be modified.
- hw2 of a 32-bit B (immediate) instruction without a condition field can be concurrently modified to hw2 of another 32-bit B (immediate) instruction without a condition field:
 - This means that some bits of the immediate value, including the least significant bits, can be modified.

Note

- In the T32 instruction set:
 - The only encodings of BKPT and SVC are 16-bit.
 - The only encoding of BL is 32-bit.
 - The ISB instruction can be concurrently modified and executed in the A32 and A64 instruction sets, but not in the T32 instruction set.
-

For the instructions explicitly identified in this section, the architecture guarantees that, after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

The instructions to which this applies are the B, BL, NOP, BKPT, SVC, HVC, and SMC instructions.

For both instruction sets, if one thread of execution changes a conditional branch instruction to another conditional branch instruction, and the change affects both the condition field and the branch target, execution of the changed instruction by another thread of execution before the change is synchronized can lead to either:

- The old condition being associated with the new target address.

- The new condition being associated with the old target address.

These possibilities apply regardless of whether the condition, either before or after the change to the branch instruction, is the *always* condition.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. No PE must be executing an instruction when another PE is modifying that instruction.
2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

```
; Coherency example for self-modifying code
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first
; line instead of STR for a 16-bit instruction.
STR <Rt>, [Rn]
DCCMVAU Rn                ; Clean data cache by MVA to point of unification (PoU)
DSB                        ; Ensure visibility of the data stored
ICIMVAU Rn                ; Invalidate instruction cache by VA to PoU
BPIMVA Rn                  ; Invalidate branch predictor by MVA to PoU
DSB
```

Note

- The DCCMVAU operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
 - If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
-
3. In a multiprocessor system, the ICIMVAU and BPIMVA are broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, once the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

```
ISB                        ; Synchronize fetched instruction stream
```

For more information about the required synchronization operation, see [Synchronization and coherency issues between data and instruction accesses](#).

For information about the Memory Effects generated by instruction fetches, see [External visibility requirement](#).

B2.3 Definition of the memory model

This section describes observation and ordering in the memory model. It contains the following subsections:

- [Basic definitions.](#)
- [Intrinsic Dependency relations.](#)
- [Fetch-intrinsically-before.](#)
- [Dependency relations.](#)
- [Ordering relations.](#)
- [Observation relations.](#)
- [Definition of an Architecturally Allowed Execution.](#)
- [Alternative formulations of the External visibility requirement for Explicit Memory Effects.](#)
- [Ordering of instruction fetches.](#)
- [Memory barriers.](#)

For more information on endpoint ordering of memory accesses, see [Reordering](#).

In the memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers, see [Memory types and attributes](#).

The architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See: [The AArch32 System Level Memory Model](#).

See also [Mismatched memory attributes](#).

B2.4 Ordering requirements defined by the formal concurrency model

The material presented in the current section corresponds to what is described in the [formal concurrency model](#). The definitions of relations, including but not limited to Dependency-ordered-before, Barrier-ordered-before, or Atomic-ordered-before, are automatically transliterated from the formal concurrency model.

The Arm memory model introduces a number of relations which are as follows:

- *Intrinsic relations*; For example, Intrinsic Data Dependencies, Intrinsic Control Dependencies, Intrinsic Order Dependencies and Translation-intrinsically-before are hardware requirements which stem from the instruction semantics.
- *After relations*; For example, Coherence-after and TLBI-after are relations which happen to be oriented that way in a specific execution, but could be oriented the other way in a different execution.
- *Observation relations*; For example, Explicit-Observed-by and TLBI-Observed-by build on *After relations* to describe an execution.
- *Ordered relations*; For example, Ordered-before and TLBI-ordered-before are architectural requirements which must be respected by hardware in all executions.

This section describes observation and ordering in the Arm memory model. It contains the following subsections:

- [Basic definitions.](#)
- [Intrinsic Dependency relations.](#)
- [Tag-check-intrinsically-before.](#)
- [Translation-intrinsically-before.](#)
- [Fetch-intrinsically-before.](#)
- [Dependency relations.](#)
- [Ordering relations.](#)
- [Observation relations.](#)
- [Definition of an Architecturally Allowed Execution.](#)
- [Alternative formulations of the External visibility requirement for Explicit Memory Effects.](#)

For more information about endpoint ordering of memory accesses, see Reordering.

In the Arm memory model, the Shareability memory attribute indicates the degree to which hardware must ensure memory coherency between a set of observers. See Memory types and attributes.

The Arm architecture defines additional memory attributes and associated behaviors, which are defined in the system level section of this manual. See:

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

See also Mismatched memory attributes.

The formal memory model described in this section covers much, but not all, of the memory system functionality. In particular the following are in scope:

- Ordering requirements for instructions that explicitly access memory (for example: LDR, STR, SWP, etc.).
 - This includes the ordering requirements between any two Explicit Memory Effects where:
 - The Explicit Memory Effects are to the same Location either from the same or different PEs.
 - One or both of the Explicit Memory Effects are generated by an atomic instruction.
 - A barrier instruction is in program order between the Explicit Memory Effects.
 - There is a dependency between the Explicit Memory Effects.
 - And excludes cases where:
 - At least one of the Explicit Memory Effects is generated by SVE instructions.
 - At least one of the Explicit Memory Effects is generated by SME instructions.
 - At least one of the Explicit Memory Effects is generated by instructions added by FEAT_MOPS.

- Ordering requirements relating to the Virtual Memory System Architecture:
 - This includes:
 - Implicit TTD Memory Read Effects.
 - Implicit TTD Memory Write Effects when FEAT_HAF or FEAT_HAFDBS is implemented.
 - MMU Fault Effects that cause Data Abort exceptions.
 - This excludes:
 - Any Effects relating to address translation instructions.
 - Any Effects relating to address translation for instruction fetch.
 - The ordering requirements between Implicit TTD Memory Effects that belong to different levels or stages of translation.
 - The ordering requirements relating to FEAT_HDBSS.
 - The ordering requirements relating to FEAT_HACDBS.
- Ordering requirements relating to tag checks when FEAT_MTE2 is implemented. This includes:
 - Implicit Tag Memory Read Effects generated by tag checked instructions.
 - Explicit Tag Memory Effects generated by tag load, tag store, tag and data store, and bulk allocation tag access instructions.
 - TagCheck Fault Effects generated by tag checked instructions.
- Ordering requirements relating to instruction fetch. This includes:
 - Implicit Instruction Memory Read Effects generated as a result of instruction fetch.
 - DC CVAU Effects.
 - IC Effects.
 - Undefined Instruction Fault Effects.

Note

Arm is in the process of formalizing many features of the memory system architecture, but this work is not complete at this time. The following items are not in scope:

- Ordering requirements relating to the Event Register.
- Ordering requirements relating to the Exclusives monitor.
- Endpoint arrival of memory accesses.
- Interactions between memory accesses and peripherals.
- Ordering requirements relating to asynchronous exceptions and error exceptions.
- Ordering requirements relating to the generic interrupt controller.
- Ordering requirements for direct and indirect System register reads and writes.
- Ordering requirements relating to FEAT_GCS.
- Ordering requirements relating to FEAT_SPE.
- Ordering requirements relating to FEAT_TRBE.
- Ordering requirements relating to FEAT_RME.
- Ordering requirements relating to RAS PE architecture.
- Ordering requirements of DC Effects generated by DC instructions other than the DC CVAU, `xt` instruction.

These will be addressed progressively in future versions of the Arm architecture, but until then, the required behaviors are described less formally in the prose and pseudocode of the Arm ARM.

For the purposes of the memory model, all Effects generated due to execution of an instruction are considered to be generated by the PE on which the instruction is executed. The list of all Effects of an instruction in the scope of the Arm Memory Model is further presented in [Effects](#).

In this section [Ordering requirements defined by the formal concurrency model](#), all of the following are assumed:

- One of the following applies:

- All PEs are part of the same Inner Shareable domain and all Memory Effects are to Inner Shareable or Outer Shareable memory as long as no TLB invalidations nor their corresponding DSB specify the Non-shareable domain.
- All PEs are part of the same Outer Shareable domain and all Memory Effects are to Outer Shareable memory as long as no TLB invalidations nor their corresponding DSB specify the Non-shareable domain or the Inner Shareable domain.
- All PEs are part of the Full System domain and all Memory Effects are to Normal, Inner Non-cacheable, Outer Non-cacheable memory or all Memory Effects are to Normal, Inner Write-Through, Outer Write-Through memory or all Memory Effects are to Device memory as long as all barriers specify the Full System domain and there are no TLB invalidations.
- All Memory Effects are to Normal, Inner Write-Back, Outer Write-Back memory. Behaviors in [Ordering requirements defined by the formal concurrency model](#) also apply to Memory Effects to memory with attributes other than Normal, Inner Write-Back, Outer Write-Back (for example, Device or Inner Write-Through, Outer Write-Through) as long as the behavior of the instruction which generated the Memory Effect is guaranteed for the memory type. For example, atomic instructions are only guaranteed to be atomic for Normal, Inner Write-Back, Outer Write-Back, Shareable memory. Memory Effects to memory with attributes other than Normal, Inner Write-Back, Outer Write-Back may require additional ordering behaviors than specified in [Ordering requirements defined by the formal concurrency model](#). Such additional behaviors are described in Memory types and attributes.
- All Memory Effects to a given Location are not using mismatched memory attributes as described in Mismatched memory attributes.

B2.4.1 Basic definitions

The Arm memory model provides a set of definitions that are used to construct conditions on the permitted sequences of accesses to memory.

Location

A *Location* is a Memory Location or a Tag Location. A *Memory Location* is a byte of data that is associated with an address in the physical address space. A *Tag Location* is a 4-bit MTE Allocation Tag that is associated with an address in the physical address space.

Effects

The *Effects* of an instruction can be:

- Register Effects.
- Memory Effects.
- Explicit Memory Effects.
- Implicit TTD Memory Effects.
- Hardware Update Effects.
- Tag Memory Effects.
- Implicit Instruction Memory Read Effects.
- Barrier Effects.
- Instruction Fetch Barrier Effects.
- Conditional Branching, and Intrinsic Branching Effects.
- Fault Effects, Exception Entry, and Exception Return Effects.
- TLBI Effects, Completed TLBI Effects and Invalidation Scopes.
- TLBUncacheable Effects.
- DC CVAU Effects.
- IC Effects, Completed IC Effects.
- Empty Effects.

Program order

An Effect E_1 of an instruction I_1 appears in program order before an Effect E_2 of an instruction I_2 if and only if I_1 occurs before I_2 in the order specified by the program. Each Effect generated by an instruction has a unique Effect identifier that characterizes it among the Effects generated by the same instruction.

Register Effects

The *Register Effects* of an instruction are register reads or register writes of that instruction. Register Effects only pertain to:

- General purpose registers from R0 to R30, excluding the zero register.
- SVE registers.
- SIMD/FP registers.
- PSTATE.NZCV.
- Direct reads from System registers that are readable.
- Direct writes to System registers that are writable.
- Direct reads from Special-purpose registers that are readable.
- Direct writes to Special-purpose registers that are writable.

For an instruction that accesses registers, a *Register Read Effect* is generated for each register read by the instruction and a *Register Write Effect* is generated for each register written by the instruction. An instruction may generate both Register Read and Write Effects.

Memory Effects

The *Memory Effects* of an instruction are the Read or Write Effects from memory of that instruction. For an instruction that accesses memory, a *Memory Read Effect* is generated for each [Location](#) read from by the instruction and a *Memory Write Effect* is generated for each [Location](#) written to by the instruction. An instruction can generate both Memory Read and Write Effects.

Explicit Memory Effects

An *Explicit Memory Effect* is a Memory Read or Memory Write Effect that is generated by one of the following:

- A load instruction, store instruction, atomic instruction, Tag and Data store instruction, DC ZVA, DC GZVA, or STZGM; to access the Memory Location(s) addressed by any of the following from the instruction:
 - The X_n | SP operand, plus any applicable offset.
 - The PC plus the offset specified by the <label> argument.
- A Tag load instruction, Tag store instruction, Tag and Data store instruction, Bulk Allocation tag access instruction, DC GZVA, or DC GVA; to access the Tag Location(s) addressed by the X_n | SP operand, plus any applicable offset, of the instruction.
- A Memory Copy or Memory Set instruction; to access the Memory Location(s) addressed by the X_s or X_d operand of the instruction, including all of the bytes within the range specified by the X_n operand of the instruction.
- A Memory Set with tag setting instruction; to access the Tag Location(s) addressed by the X_s or X_d operand of the instruction, including all of the Allocation tags within the range specified by the X_n operand of the instruction.
- An MRS or MSR instruction that is transformed into a Memory Effect due to FEAT_NV2; to access the Memory Locations addressed by VNCR_EL2 plus the offset for the specified register.

Note

Prefetch memory instructions do not generate any Explicit Memory Effects.

Implicit TTD Memory Effects

Implicit Translation Table Descriptor Memory Effects (Implicit TTD Memory Effects) are Memory Read or Write Effects to translation table descriptors caused by address translation, including the hardware updates for the Access Flag or the Dirty State of a TTD.

Hardware Update Effects

Hardware Update Effects are Implicit TTD Memory Write Effects that correspond to writes modifying the Access Flag or the Dirty State of a TTD.

For more information on Hardware Updates of the Access Flag or Dirty State, see Hardware management of the Access flag and Hardware management of the dirty state.

Note

A Hardware Update Effect for the Access Flag may occur even in the absence of corresponding architecturally executed instructions, in which case the Hardware Update Effect is considered autonomous. Autonomous Hardware Update Effects are not constrained by the ordering relations that apply to the Hardware Update Effects of architecturally executed instructions.

Tag Memory Effects

Tag Memory Effects are a subset of Memory Effects. A Tag Memory Effect refers to a Memory Read or Write Effect to a [Tag Location](#). For example:

- LDG and LDGM instructions generate an Explicit Tag Memory Read Effect for their read from a [Tag Location](#).
- STG, STGM, STGP, DC GZVA, and DC GVA instructions generate an Explicit Tag Memory Write Effect for their write to a [Tag Location](#).
- A Tag Checked memory access generates an Implicit Tag Memory Read Effect for its read of a [Tag Location](#), as well as an Explicit Data Memory Effect.

Implicit Instruction Memory Read Effects

An *Implicit Instruction Memory Read Effect* refers to a Memory Read Effect generated as a result of an instruction fetch.

Barrier Effects

Barrier instructions generate *Barrier Effects*. In this chapter, the resulting Effect is named after the instruction. Therefore, an ISB instruction generates an ISB Effect, a DMB instruction generates a DMB Effect, and a DSB instruction generates a DSB Effect.

The following conventions are used when referring to DMB Effects:

- A DMB FULL Effect is generated by a DMB instruction with neither a LD nor ST qualifier.
- A DMB LD Effect is generated by a DMB instruction with the LD qualifier.
- A DMB ST Effect is generated by a DMB instruction with the ST qualifier.

The following conventions are used when referring to DSB Effects:

- A DSB FULL Effect is generated by a DSB instruction with neither a LD nor ST qualifier.
- A DSB LD Effect is generated by a DSB instruction with the LD qualifier.
- A DSB ST Effect is generated by a DSB instruction with the ST qualifier.

Instruction Fetch Barrier Effects

An *Instruction Fetch Barrier Effect* (IFBE) is generated by any of the following:

- A [Context Synchronization event](#).
- An exception entry to ELx, regardless of the value of SCTLR_ELx.EIS, due to an exception generated for any reason other than any of the following:
 - SVC instruction execution that is not trapped.
 - HVC instruction execution that is not disabled.
 - SMC instruction execution that is not trapped or disabled.
 - [BKPT](#) instruction execution.
 - BRK instruction execution.

Note

- If FEAT_ExS is not implemented, or if FEAT_ExS is implemented and the SCTLR_ELx.EOS field is 1, an Exception Return effect is an IFBE due to it being a [Context Synchronization event](#).
 - If FEAT_ExS is not implemented, or if FEAT_ExS is implemented and the SCTLR_ELx.EIS field is 1, the Exception entry effects due to the exceptions excluded above are IFBEs due to being [Context Synchronization events](#).
-

Conditional Branching, and Intrinsic Branching Effects

Branching Effects are generated when a choice is made between two possible paths in the execution flow. Branching Effects represent such decision points.

Conditional Branching Effects are generated by Conditional branch instructions, when as a consequence of evaluating a condition a choice potentially affecting the execution flow of other subsequent instructions is made.

Intrinsic Branching Effects due to a translation occur when, as a consequence of reading a TTD, a choice is made between raising a fault (for example, if the descriptor is invalid) or performing a physical access. *Intrinsic Branching Effects* represent such decision points.

Intrinsic Branching Effects due to tag checking occur when, as a consequence of reading a tag, a choice is made between raising a fault (that is, due to a failed tag check) or performing a physical access. *Intrinsic Branching Effects* represent such decision points.

Intrinsic Branching Effects due to instruction fetch occur when, as a consequence of reading and decoding an instruction, a choice is made between raising a fault (for example, as a result of fetching an illegal instruction) or proceeding to execute the valid instruction. *Intrinsic Branching Effects* represent such decision points.

Note

Conditional Selection instructions and Compare-and-Swap instructions generate *Intrinsic Branching Effects*.

Fault Effects, Exception Entry, and Exception Return Effects

An instruction that encounters a [fault](#) generates a Fault Effect. An instruction that encounters a fault leading to a synchronous exception generates a Fault Effect that is also an *Exception Entry Effect*, for example as a result of reading an invalid TTD or fetching an illegal instruction.

An ERET instruction generates an *Exception Return Effect*.

A Fault Effect generated by the MMU is called an *MMU Fault Effect*. See MMU fault types for a full list of faults generated by the MMU.

A Fault Effect generated by a failed Tag Check is called a *TagCheck Fault Effect*.

Note

- An instruction that encounters an asynchronous Tag Check Fault generates a TagCheck Fault Effect that is not an Exception Entry Effect.
 - An instruction that encounters a synchronous Tag Check Fault generates a TagCheck Fault Effect that is also an Exception Entry Effect.
-

TLBI Effects, Completed TLBI Effects and Invalidation Scopes

TLBI Effects are generated by TLBI instructions.

A TLBI Effect E_1 is a *Completed TLBI Effect* if E_1 appears in program order before a DSB FULL Effect.

The *Invalidation scope* of a TLBI Effect is a set Memory Effects. For a Virtual Address x , a Memory Read or Write Effect to the TTD of x is in the Invalidation Scope of a TLBI Effect if x is in the scope of the TLBI instruction. For the full description of the scope of TLBI instructions, see TLB maintenance instruction scope.

TLBUncacheable Effects

An MMU Fault Effect is TLBUncacheable if it is one of:

- A Translation Fault Effect.
- An Access Flag Fault Effect.
- An Address Size Fault Effect.
- A synchronous External Abort on translation table walk, such that it cannot be determined whether the TTD would otherwise generate a Translation Fault, Access Flag Fault, or Address Size Fault effect.

Note

A Permission Fault Effect is not TLBUncacheable.

DC CVAU Effects

DC CVAU Effects are generated by DC CVAU instructions.

IC Effects, Completed IC Effects

IC Effects are generated by IC IALLUIS, IC IALLU or IC IVAU instructions. An IC Effect E_1 is a *Completed IC Effect* if E_1 appears in program order before a DSB FULL Effect.

Empty Effects

Empty Effects are generated by instructions where a decision point, represented by a Branching Effect, leads to no further action resulting from that decision. For example, when FEAT_MTE_ASYNC is implemented and Tag Check Faults are reported asynchronously, a successful tag comparison generates an Empty Effect instead of a TagCheck Fault Effect.

Low-order Bits

The Low-order Bits of an address are the bits that remain unchanged in the translation process of an input address to an output address. The number of low-order bits that remain unchanged depends on the translation granule size.

Same Low-order Bits

Two Effects E_1 and E_2 have the Same Low-order Bits if the Low-order Bits of the address of E_1 and the Low-order Bits of the address of E_2 match.

Note

The address associated with an MMU Fault Effect is the faulting address as indicated in FAR_ELx.

Effects with a valid Physical Address

An Effect E_1 is an Effect with a valid Physical Address if one of the following applies:

- E_1 is a Memory Effect.
- E_1 is a DC Effect.
- E_1 is an IC Effect.

Note

In situations where accessing a Physical Address generates an MMU fault, the PE does not generate the corresponding Memory Effect or DC Effect or IC Effect.

Same Location

An Effect E_1 and an Effect E_2 are to the *Same Location* if one of the following applies:

- All of the following apply:
 - E_1 is an Effect with a valid Physical Address.
 - It is not the case that E_1 is a Tag Memory Effect.
 - E_1 and E_2 are to the same Physical Address.
 - E_2 is an Effect with a valid Physical Address.
 - It is not the case that E_2 is a Tag Memory Effect.
- All of the following apply:
 - E_1 is a Tag Memory Effect.
 - E_1 and E_2 are to the same Physical Address.
 - E_2 is a Tag Memory Effect.

Same Cache Line

An Effect E_1 and an Effect E_2 are to the *Same Cache Line* if:

- E_1 and E_2 are Memory Effects or DC CVAU Effects or IC Effects.
- E_1 and E_2 are to PAs that are on the Same Cache Line as follows from section Cache identification.

Successful Read-Modify-Write pair

Two Effects E_1 and E_2 form a *Successful Read-Modify-Write* pair if all of the following apply:

- E_1 is a Memory Read Effect.
- E_2 is a Memory Write Effect.
- E_1 and E_2 are to the Same Location.
- One of the following applies:
 - E_1 and E_2 are generated by the same atomic instruction.
 - E_1 and E_2 are generated by the same successful Load-Exclusive/Store-Exclusive instruction pair.

Reads-from-memory

The *Reads-from-memory* relation couples Memory Read and Write Effects to the same [Location](#) so that each Memory Read Effect is paired with exactly one Memory Write Effect in the execution of a program. A Memory Read Effect E_2 Reads-from-memory a Memory Write Effect E_1 , if and only if E_1 and E_2 are to the same location and E_2 takes its data from E_1 .

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2 .
- E_1 and E_2 are to the Same Location.

then it is not the case that E_1 Reads-from-memory E_2 .

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 and E_2 form a successful Read-Modify-Write pair.

then it is not the case that E_1 Reads-from-memory E_2 .

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Implicit Tag Memory Read Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2 .
- E_1 and E_2 are to the Same Location.

then it is not the case that E_1 Reads-from-memory E_2 .

Coherence order

There is a per-location *Coherence order* relation that provides a total order over all Memory Write Effects to that Location, starting with a notional Memory Write Effect of the initial value. The Coherence order of a [Location](#) represents the order in which Memory Write Effects to the [Location](#) arrive at memory.

Local register read successor

An Effect E_2 is a *Local register read successor* of an Effect E_1 if all of the following apply:

- E_1 is a Register Write Effect.
- E_1 appears in program order before E_2 .
- E_1 and E_2 are to the same Register.
- It is not the case that all of the following apply:
 - E_1 appears in program order before E_3 .

- E_1 and E_3 are to the same Register.
- E_3 is a Register Write Effect.
- E_3 appears in program order before E_2 .
- E_3 and E_2 are to the same Register.
- E_2 is a Register Read Effect.

Local memory read successor

An Effect E_2 is a *Local memory read successor* of an Effect E_1 if all of the following apply:

- E_1 is a Memory Write Effect.
- E_1 appears in program order before E_2 .
- E_1 and E_2 are to the Same Location.
- E_2 is a Memory Read Effect.
- There is no Memory Write Effect E_3 to the Same Location as E_1 such that E_1 appears in program order before E_3 and E_3 appears in program order before E_2 .

Local memory write successor

An Effect E_2 is a *Local memory write successor* of an Effect E_1 if one of the following applies:

- All of the following apply:
 - One of the following applies:
 - E_1 is an Explicit Memory Effect.
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 appears in program order before E_2 .
 - E_1 and E_2 are to the Same Location.
 - E_2 is an Explicit Memory Write Effect.
- All of the following apply:
 - E_1 is an Implicit TTD Memory Read Effect.
 - E_1 appears in program order before E_2 .
 - E_1 and E_2 are to the Same Location.
 - One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.

Coherence-before, Coherence-after

A Memory Write Effect E_1 is *Coherence-before* a Memory Write Effect E_2 to the same [Location](#) if E_1 is sequenced before E_2 in the Coherence order for the [Location](#).

A Memory Read Effect E_1 is *Coherence-before* a Memory Write Effect E_2 to the same [Location](#) if E_1 Reads-from-memory a Memory Write Effect E_3 and E_3 is Coherence-before E_2 .

An Effect E_2 is *Coherence-after* an Effect E_1 if E_1 is *Coherence-before* E_2 .

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 appears in program order before E_2 .
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1 .

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Explicit Memory Read Effect.
- E_1 appears in program order before E_2 .
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1 .

For two Effects E_1 and E_2 if all of the following apply:

- E_1 is an Explicit Memory Write Effect.
- E_2 is an Implicit Tag Memory Read Effect.
- E_1 appears in program order before E_2 .
- E_1 and E_2 are to the Same Location.

then it is not the case that E_2 is Coherence-before E_1 .

TLBI-before, TLBI-after

For two Effects E_1 and E_2 , if and only if all of the following applies:

- E_1 is a Completed TLBI Effect.
- E_2 is an Implicit TTD Memory Read Effect.
- E_2 is in the Invalidation Scope of E_1 .

then one and only one of the following applies:

- E_1 is *TLBI-after* E_2 (equivalently E_2 is *TLBI-before* E_1), or
- E_2 is *TLBI-after* E_1 (equivalently E_1 is *TLBI-before* E_2).

Note

The TLBI-after relation enumerates all possible pairs (E_1, E_2) where E_1 is a TLBI Effect and E_2 is an Implicit TTD Memory Read Effect such that E_2 is in the Invalidation Scope of E_1 , and the TLBI-after relation is asymmetric.

DC-before, DC-after

For two Effects E_1 and E_2 , if all of the following applies:

- E_1 is a DC CVAU Effect.
- E_2 is an Explicit Memory Write Effect.
- E_1 and E_2 are to the Same Cache Line.

then one of the following applies:

- E_1 is *DC-after* E_2 (equivalently E_2 is *DC-before* E_1), or
- E_2 is *DC-after* E_1 (equivalently E_1 is *DC-before* E_2).

Note

The DC-after relation enumerates all possible pairs (E_1, E_2) where E_1 is a DC CVAU Effect and E_2 is an Explicit Memory Write Effect such that E_1 and E_2 are to the Same Cache Line, and the relation DC-after is asymmetric.

IC-before, IC-after

For two Effects E_1 and E_2 , if and only if all of the following applies:

- E_1 is a completed IC Effect.
- E_2 is an Implicit Instruction Memory Read Effect.
- E_1 and E_2 are to the Same Cache Line.

then one and only one of the following applies:

- E_1 is *IC-after* E_2 (equivalently E_2 is *IC-before* E_1), or
- E_2 is *IC-after* E_1 (equivalently E_1 is *IC-before* E_2).

Note

The IC-after relation enumerates all possible pairs (E_1, E_2) where E_1 is a completed IC Effect and E_2 is an Implicit Instruction Memory Read Effect such that E_1 and E_2 are to the Same Cache Line, and the relation IC-after is asymmetric.

Single-copy-atomic grouping

A *single-copy-atomic grouping* gathers Explicit Memory Effects generated by the same instruction when the architecture requires the Explicit Memory Effects to be single-copy atomic as defined in Requirements for single-copy atomicity. For example, an aligned STRH instruction generates two Explicit Memory Write Effects that are in the same single-copy-atomic grouping.

Atomicity of Successful Read-Modify-Write pair

For two Effects E_1 and E_2 if all of the following apply:

- E_1 and E_2 form a successful Read-Modify-Write pair.
- There is an Explicit Memory Write Effect E_3 .
- It is not the case that all of the following apply:
 - E_1 is an Explicit Memory Effect.
 - E_1 and E_3 are from the same Processing Element.
 - E_2 is an Explicit Memory Effect.
 - E_2 and E_3 are from the same Processing Element.

then it is not the case that E_1 is Coherence-before E_3 and E_3 is Coherence-before E_2 .

B2.4.2 Intrinsic Dependency relations

Intrinsic Dependencies are as follows:

Intrinsic Data Dependency

There is an *Intrinsic Data Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- It is not the case that E_1 is an Intrinsic Branching Effect.
- E_1 and E_2 are generated by the same instruction $\mathbb{I}$.
- The instruction semantics for that instruction $\mathbb{I}$ indicate that the value produced by E_1 is consumed by E_2 .

Note

This applies even in cases where the Intrinsic Data Dependency might be perceived to be a false dependency.

Intrinsic Control Dependency

There is an *Intrinsic Control Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- E_1 is an Intrinsic Branching Effect.
- E_1 and E_2 are generated by the same instruction $\mathbb{I}$.
- The instruction semantics for that instruction $\mathbb{I}$ indicate that the generation of E_2 is conditional and based on the determination of E_1 .

Intrinsic Order Dependency

There is an *Intrinsic Order Dependency* from an Effect E_1 to an Effect E_2 if and only if all the following apply:

- E_1 is a Memory Effect.
- E_1 and E_2 are generated by the same instruction $\mathbb{I}$.
- The instruction semantics for that instruction $\mathbb{I}$ indicate that E_1 is generated before E_2 .
- It is not the case that there is an Intrinsic Data Dependency from E_1 to E_2 .
- It is not the case that there is an Intrinsic Control Dependency from E_1 to E_2 .
- E_2 is a Memory Effect.

In the following, instructions whose Intrinsic dependencies are special cases that arose during the formalization of the memory model are listed. This does not preclude the existence of other special cases in the future.

B2.4.2.1 Conditional Selection instructions

The following applies to the Effects generated by `CSEL Xd,Xn,Xm, cond`

When the condition `cond` is true all of the following apply:

- Dt_1 : There is an Intrinsic Data Dependency from the Register Read Effect of `PSTATE.NZCV` to the Intrinsic Branching Effect corresponding to the condition `cond`.
- Ct_1 : There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks the condition `cond` to the Register Read Effect of X_n .
- Dt_2 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Register Write Effect of X_d .

When the condition `cond` is false all of the following apply:

- Df_1 : There is an Intrinsic Data Dependency from the Register Read Effect of `PSTATE.NZCV` to the Intrinsic Branching Effect corresponding to the condition `cond`.
- Cf_1 : There is an Intrinsic Control Dependency from the Intrinsic Branching Effect corresponding to the condition `cond` to the Register Read Effect of X_m .
- Df_1 : There is an Intrinsic Data Dependency from the Register Read Effect of X_m to the Register Write Effect of X_d .

B2.4.2.2 Compare-and-Swap instructions

The following apply to the Effects generated by `CAS Xs,Xt,[Xn]`:

When the value of the Memory Location x addressed by X_n is not equal to the value of X_s one of the following applies:

- D_1 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n .
- D_2 : There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s .
- D_3 : There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s .
- Df_1 : There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_s .

When the value of the Memory Location x addressed by X_n is equal to the value of X_s one of the following applies:

- All of the following apply:
 - D_1 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n .
 - D_2 : There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s .
 - D_3 : There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s .
 - DS_1 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .
 - DS_2 : There is an Intrinsic Data Dependency from Register Read Effect X_t to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .
 - CS_1 : There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .
 - DS_{31} : There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_s .
- All of the following apply:

- D_1 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n .
- D_2 : There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s .
- D_3 : There is an Intrinsic Data Dependency from the Register Read Effect of X_s to the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s .
- DS_1 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .
- DS_2 : There is an Intrinsic Data Dependency from the Register Read Effect of X_t to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .
- CS_1 : There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .
- DS_3 : There is an Intrinsic Data Dependency from Register Read Effect X_s to the Register Write Effect of X_s .
- CS_2 : There is an Intrinsic Control Dependency from the Intrinsic Branching Effect which checks whether the contents of the Memory Location x are equal to the contents of the Register X_s to the Register Write Effect of X_s .

B2.4.2.3 Swap instructions

All of the following apply to the Effects generated by the `SWP  $X_s, X_t, [X_n]$` instruction:

- D_1 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Read Effect of the Memory Location x addressed by X_n .
- D_2 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .
- O_1 : There is an Intrinsic Order Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .
- D_3 : There is an Intrinsic Data Dependency from the Explicit Memory Read Effect of the Memory Location x addressed by X_n to the Register Write Effect of X_t .
- D_4 : There is an Intrinsic Data Dependency from the Register Read Effect of X_s , to the Explicit Memory Write Effect of the Memory Location x addressed by X_n .

B2.4.2.4 Branch with Link to Register instructions

All of the following apply to the Effects generated by the `BLR  $X_n$` instruction:

- D_1 : There is an Intrinsic Data Dependency from the Register Read Effect of X_n to the Branching Effect of the instruction.

B2.4.3 Fetch-intrinsically-before

An Effect E_1 is *Fetch-intrinsically-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- There is an Intrinsic Data Dependency from E_1 to E_3 .
- E_3 is a Branching Effect.
- One of the following applies:
 - There is an Intrinsic Control Dependency from E_3 to E_2 .
 - All of the following apply:
 - There is an Intrinsic Control Dependency from E_3 to E_4 .
 - There is an Intrinsic Data Dependency from E_4 to E_2 .

B2.4.4 Dependency relations

Dependency through registers and memory

There is a *Dependency through registers and memory* from an Effect E_1 to an Effect E_2 if one of the following applies:

- All of the following apply:
 - It is not the case that E_1 is generated by a Store-Exclusive instruction as part of a successful Load-Exclusive/Store-Exclusive pair.
 - E_2 is a Local register read successor of E_1 .
- E_2 is a Local memory read successor of E_1 .
- There is an Intrinsic Data Dependency from E_1 to E_2 .
- All of the following apply:
 - There is a Dependency through registers and memory from E_1 to E_3 .
 - There is a Dependency through registers and memory from E_3 to E_2 .

Data dependency

There is a *Data dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3 .
- E_1 appears in program order before E_3 .
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2 .
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the data value written by E_2 .
- E_2 is an Explicit Memory Write Effect.

Address dependency

There is an *Address dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3 .
- E_1 appears in program order before E_3 .
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2 .
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the address of the Location of E_2 .
- One of the following applies:
 - E_2 is an Explicit Memory Effect.
 - E_2 is an Implicit Tag Memory Read Effect.
 - E_2 is an Implicit TTD Memory Read Effect.
 - E_2 is a Hardware Update Effect.
 - E_2 is a TLBI Effect.
 - E_2 is a DC CVAU Effect.
 - E_2 is an IC IVAU Effect.

Control dependency

There is a *Control dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Dependency through registers and memory from E_1 to E_3 .
- E_1 appears in program order before E_3 .
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_4 .
- E_4 is a Conditional Branching Effect.
- E_4 appears in program order before E_2 .

Pick dependency through registers and memory

There is a *Pick dependency through registers and memory* from an Effect E_1 to an Effect E_2 if one of the following applies:

- There is a Dependency through registers and memory from E_1 to E_2 .
- There is an Intrinsic Control Dependency from E_1 to E_2 .
- All of the following apply:
 - There is a Pick dependency through registers and memory from E_1 to E_3 .
 - There is a Pick dependency through registers and memory from E_3 to E_2 .

Pick Basic dependency

There is a *Pick Basic dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_2 .

Pick Data dependency

There is a *Pick Data dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3 .
- E_1 appears in program order before E_3 .
- E_3 is a Register Read Effect.
- One of the following applies:
 - There is an Intrinsic Data Dependency from E_3 to E_2 .
 - All of the following apply:
 - There is an Intrinsic Data Dependency from E_3 to E_4 .
 - There is an Intrinsic Control Dependency from E_4 to E_2 .
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the data value written by E_2 .
- E_2 is an Explicit Memory Write Effect.

Pick Address dependency

There is a *Pick Address dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3 .
- E_1 appears in program order before E_3 .
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_2 .
- The semantics of the instruction that generates E_3 and E_2 indicate that the value produced by E_3 is consumed in the calculation of the address of the Location of E_2 .
- One of the following applies:
 - E_2 is an Explicit Memory Effect.
 - E_2 is an Implicit Tag Memory Read Effect.
 - E_2 is an Implicit TTD Memory Read Effect.
 - E_2 is a Hardware Update Effect.
 - E_2 is a TLBI Effect.
 - E_2 is a DC CVAU Effect.
 - E_2 is an IC IVAU Effect.

Pick Control dependency

There is a *Pick Control dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- There is a Pick dependency through registers and memory from E_1 to E_3 .

- E_1 appears in program order before E_3 .
- E_3 is a Register Read Effect.
- There is an Intrinsic Data Dependency from E_3 to E_4 .
- E_4 is a Conditional Branching Effect.
- E_4 appears in program order before E_2 .

Pick dependency

There is a *Pick dependency* from an Effect E_1 to an Effect E_2 if all of the following apply:

- One of the following applies:
 - There is a Pick Basic dependency from E_1 to E_2 .
 - There is a Pick Address dependency from E_1 to E_2 .
 - There is a Pick Data dependency from E_1 to E_2 .
 - There is a Pick Control dependency from E_1 to E_2 .
- It is not the case that E_1 and E_2 are generated by the same instruction.

B2.4.5 Ordering relations

Explicit-hazard-ordered-before

An Effect E_1 is *Explicit-hazard-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_1 appears in program order before E_3 .
- E_1 and E_3 are to the Same Location.
- E_3 is an Explicit Memory Read Effect.
- One of the following applies:
 - E_3 belongs to the same single-copy-atomic grouping as E_4 .
 - E_3 and E_4 are the same Effect.
- E_4 is an Explicit Memory Read Effect.
- E_4 is Coherence-before E_5 .
- E_4 and E_5 are from different Processing Elements.
- One of the following applies:
 - E_5 is an Explicit Memory Write Effect.
 - E_5 is a Hardware Update Effect.
- One of the following applies:
 - E_5 belongs to the same single-copy-atomic grouping as E_2 .
 - E_5 and E_2 are the same Effect.
- One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.

Note

The Explicit-hazard-ordered-before relation does not apply when either E_1 or E_2 or both are generated by SVE instructions.

TTD-read-ordered-before

An Effect E_1 is *TTD-read-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_1 is TLBI-before E_3 .
 - E_3 is a TLBI Effect.
 - E_3 is DSB-ordered-before E_2 .
- All of the following apply:
 - E_1 is TLBI-before E_3 .

- E_3 is a TLBI Effect.
- E_3 is Instruction-fetch-barrier-ordered-before E_2 .

TLBI-ordered-before

An Effect E_1 is *TLBI-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is TTD-read-ordered-before E_2 .
- All of the following apply:
 - E_3 is Translation-intrinsically-before E_1 .
 - E_3 is TTD-read-ordered-before E_2 .
 - E_3 and E_2 are from different Processing Elements.
- All of the following apply:
 - E_1 is an Implicit TTD Memory Read Effect.
 - E_1 is Translation-intrinsically-before E_4 .
 - E_4 and E_5 have the Same Low-order Bits.
 - E_4 appears in program order before E_5 .
 - E_3 is Translation-intrinsically-before E_5 .
 - E_3 is an Implicit TTD Memory Read Effect.
 - E_1 and E_3 are generated in the same translation context.
 - E_3 is TTD-read-ordered-before E_2 .
 - E_3 and E_2 are from different Processing Elements.

Note

The clause of TLBI-ordered-before which refers to “Same Low-order Bits” applies in cases where an instruction I_1 which generates E_1 and E_4 appears in program order before an instruction I_2 which generates E_3 and E_5 , and I_1 and I_2 access overlapping bytes in memory.

Instruction-read-ordered-before

An Effect E_1 is *Instruction-read-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_1 is IC-before E_3 .
 - E_3 is an IC Effect.
 - E_3 is DSB-ordered-before E_2 .
- All of the following apply:
 - E_1 is IC-before E_3 .
 - E_3 is an IC Effect.
 - E_3 is Instruction-fetch-barrier-ordered-before E_2 .
- All of the following apply:
 - The [Effective value](#) of CTR_EL0.DIC is 1.
 - E_1 is Coherence-before E_2 .

IC-ordered-before

An Effect E_1 is *IC-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- E_1 appears in program order before E_3 .
- E_3 is an Implicit Instruction Memory Read Effect.
- E_3 is Instruction-read-ordered-before E_2 .

Hazard-ordered-before

An Effect E_1 is *Hazard-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Explicit-hazard-ordered-before E_2 .
- E_1 is TLBI-ordered-before E_2 .
- E_1 is IC-ordered-before E_2 .

ETS-ordered-before

An Effect E_1 is *ETS-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - FEAT_ETSS2 is implemented.
 - E_1 is an Explicit Memory Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a TLBUncacheable Fault Effect.
 - E_2 is Translation-intrinsically-before E_3 .
 - E_2 is an Implicit TTD Memory Read Effect.
- All of the following apply:
 - FEAT_ETSS3 is implemented.
 - E_1 is an Explicit Memory Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is an MMU Fault Effect.
 - E_2 is Translation-intrinsically-before E_3 .
 - E_2 is an Implicit TTD Memory Read Effect.
- All of the following apply:
 - FEAT_ETSS3 is implemented.
 - E_1 is an Explicit Memory Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a TagCheck Fault Effect that generates a synchronous exception.
 - E_2 is Tag-check-intrinsically-before E_3 .
 - E_2 is an Implicit Tag Memory Read Effect.

Note

The ETS-ordered-before relation does not apply when E_1 is generated by an Advanced SIMD, floating-point, SVE, or SME instruction. It also does not apply when E_2 is an Implicit TTD Memory Read Effect generated for an instruction fetch.

Fetch-ordered-before

An Effect E_1 is *Fetch-ordered-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit Instruction Memory Read Effect.
- E_1 appears in program order before E_2 .
- It is not the case that E_2 is an Implicit Instruction Memory Read Effect.

Same-Cache-Line-ordered-before

An Effect E_1 is *Same-Cache-Line-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_1 is an Explicit Memory Effect.
 - E_1 appears in program order before E_2 .
 - E_1 and E_2 are to the Same Cache Line.
 - E_2 is a DC CVAU Effect.
- All of the following apply:
 - E_1 is a DC CVAU Effect.
 - E_1 appears in program order before E_2 .
 - E_1 and E_2 are to the Same Cache Line.
 - E_2 is an Explicit Memory Effect.
- All of the following apply:
 - E_1 is a DC CVAU Effect.
 - E_1 appears in program order before E_2 .
 - E_1 and E_2 are to the Same Cache Line.
 - E_2 is a DC CVAU Effect.

DSB-ordered-before

An Effect E_1 is *DSB-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - One of the following applies:
 - E_1 is a Memory Effect.
 - E_1 is a DC CVAU Effect.
 - E_1 is an IC Effect.
 - E_1 is a TLBI Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DSB FULL Effect.
 - E_3 appears in program order before E_2 .
 - It is not the case that one of the following applies:
 - E_2 is an Implicit TTD Memory Effect.
 - E_2 is an Implicit Instruction Memory Read Effect.
- All of the following apply:
 - FEAT_ETS2 is implemented.
 - One of the following applies:
 - E_1 is a Memory Effect.
 - E_1 is a DC CVAU Effect.
 - E_1 is an IC Effect.
 - E_1 is a TLBI Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DSB FULL Effect.
 - E_3 appears in program order before E_2 .
 - E_2 is an Implicit TTD Memory Effect.
- All of the following apply:
 - One of the following applies:
 - All of the following apply:
 - E_1 is an Explicit Memory Read Effect.
 - It is not the case that E_1 is generated by an atomic instruction whose destination register is WZR or XZR.
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DSB LD Effect.
 - E_3 appears in program order before E_2 .
 - It is not the case that one of the following applies:
 - E_2 is an Implicit TTD Memory Effect.
 - E_2 is an Implicit Instruction Memory Read Effect.
- All of the following apply:
 - FEAT_ETS2 is implemented.
 - E_1 is an Explicit Memory Read Effect.
 - It is not the case that E_1 is generated by an atomic instruction whose destination register is WZR or XZR.
 - E_1 appears in program order before E_3 .
 - E_3 is a DSB LD Effect.
 - E_3 appears in program order before E_2 .
 - E_2 is an Implicit TTD Memory Effect.
- All of the following apply:
 - E_1 is an Explicit Memory Write Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DSB ST Effect.
 - E_3 appears in program order before E_2 .
 - It is not the case that one of the following applies:

- E_2 is an Implicit TTD Memory Effect.
- E_2 is an Implicit Instruction Memory Read Effect.
- All of the following apply:
 - FEAT_ETS2 is implemented.
 - E_1 is an Explicit Memory Write Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DSB ST Effect.
 - E_3 appears in program order before E_2 .
 - E_2 is an Implicit TTD Memory Effect.

Note

Regardless of whether FEAT_ETS2 is implemented, the DSB-ordered-before relation does not apply when E_2 is an Implicit TTD Memory Read Effect generated for an instruction fetch.

Instruction-fetch-barrier-ordered-before

An Effect E_1 is *Instruction-fetch-barrier-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_1 is an Explicit Memory Read Effect.
 - There is a Control dependency from E_1 to E_3 .
 - E_3 is an Instruction Fetch Barrier Effect.
 - E_3 appears in program order before E_2 .
- All of the following apply:
 - E_1 is an Explicit Memory Read Effect.
 - There is a Pick Control dependency from E_1 to E_3 .
 - E_3 is an Instruction Fetch Barrier Effect.
 - E_3 appears in program order before E_2 .
- All of the following apply:
 - E_1 is an Explicit Memory Read Effect.
 - There is an Address dependency from E_1 to E_3 .
 - E_3 is an Explicit Memory Effect.
 - E_3 appears in program order before E_4 .
 - E_4 is an Instruction Fetch Barrier Effect.
 - E_4 appears in program order before E_2 .
- All of the following apply:
 - E_1 is an Explicit Memory Read Effect.
 - There is a Pick Address dependency from E_1 to E_3 .
 - E_3 is an Explicit Memory Effect.
 - E_3 appears in program order before E_4 .
 - E_4 is an Instruction Fetch Barrier Effect.
 - E_4 appears in program order before E_2 .
- All of the following apply:
 - E_1 is an Explicit Memory Read Effect.
 - There is a Pick Address dependency from E_1 to E_3 .
 - One of the following applies:
 - E_3 is Tag-check-intrinsically-before E_4 .
 - E_3 is Translation-intrinsically-before E_4 .
 - E_4 is an Instruction Fetch Barrier Effect.
 - E_4 appears in program order before E_2 .
- All of the following apply:
 - E_1 is DSB-ordered-before E_3 .
 - E_3 is an Instruction Fetch Barrier Effect.
 - E_3 appears in program order before E_2 .
- All of the following apply:

- E_1 is an Implicit TTD Memory Read Effect.
- E_1 is Translation-intrinsically-before E_3 .
- E_3 is an Instruction Fetch Barrier Effect.
- E_3 appears in program order before E_2 .
- All of the following apply:
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 is Tag-check-intrinsically-before E_3 .
 - E_3 is an Instruction Fetch Barrier Effect.
 - E_3 appears in program order before E_2 .
- All of the following apply:
 - E_1 is an Implicit TTD Memory Read Effect.
 - E_1 is Translation-intrinsically-before E_3 .
 - E_3 is an Explicit Memory Effect.
 - E_3 appears in program order before E_4 .
 - E_4 is an Instruction Fetch Barrier Effect.
 - E_4 appears in program order before E_2 .
- All of the following apply:
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 is Tag-check-before E_3 .
 - E_3 is an Explicit Memory Effect.
 - E_3 appears in program order before E_4 .
 - E_4 is an Instruction Fetch Barrier Effect.
 - E_4 appears in program order before E_2 .

Dependency-ordered-before

An Effect E_1 is *Dependency-ordered-before* an Effect E_2 if one of the following applies:

- There is an Address dependency from E_1 to E_2 .
- There is a Data dependency from E_1 to E_2 .
- All of the following apply:
 - There is a Control dependency from E_1 to E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.
 - E_2 is a TLBI Effect.
 - E_2 is a DC CVAU Effect.
 - E_2 is an IC Effect.
- All of the following apply:
 - There is an Address dependency from E_1 to E_3 .
 - E_3 is an Explicit Memory Effect.
 - E_3 appears in program order before E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.
- All of the following apply:
 - There is an Address dependency from E_1 to E_3 .
 - E_3 is an Explicit Memory Effect.
 - E_2 is a Local memory read successor of E_3 .
 - One of the following applies:
 - E_2 is an Explicit Memory Read Effect.
 - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
 - There is a Data dependency from E_1 to E_3 .
 - E_3 is an Explicit Memory Effect.

- E_2 is a Local memory read successor of E_3 .
- One of the following applies:
 - E_2 is an Explicit Memory Read Effect.
 - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
 - E_1 is an Implicit TTD Memory Read Effect.
 - E_1 is Translation-intrinsically-before E_3 .
 - E_3 is an Explicit Memory Effect.
 - E_3 appears in program order before E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.
- All of the following apply:
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 is Tag-check-before E_3 .
 - E_3 is an Explicit Memory Effect.
 - E_3 appears in program order before E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.

Intrinsically-ordered-before

An Effect E_1 is *Intrinsically-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Tag-check-intrinsically-before E_2 .
- E_1 is Translation-intrinsically-before E_2 .
- E_1 is Fetch-intrinsically-before E_2 .
- All of the following apply:
 - E_1 is an Explicit Memory Effect.
 - There is an Intrinsic Order Dependency from E_1 to E_2 .
 - E_2 is an Explicit Memory Effect.

Pick-ordered-before

An Effect E_1 is *Pick-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - There is a Pick Address dependency from E_1 to E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.
 - E_2 is a TLBI Effect.
 - E_2 is a DC CVAU Effect.
 - E_2 is an IC Effect.
- There is a Pick Data dependency from E_1 to E_2 .
- All of the following apply:
 - There is a Pick Control dependency from E_1 to E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.
 - E_2 is a TLBI Effect.
 - E_2 is a DC CVAU Effect.
 - E_2 is an IC Effect.
- All of the following apply:
 - There is a Pick Address dependency from E_1 to E_3 .
 - E_3 is an Explicit Memory Effect.

- E_3 appears in program order before E_2 .
- One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.

Atomic-ordered-before

E_1 and E_2 form a successful Read-Modify-Write pair if and only if E_1 and E_2 are generated by the same atomic instruction or the same successful Load-Exclusive/Store-Exclusive instruction pair to the Same Location.

An Effect E_1 is *Atomic-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_1 is an Explicit Memory Effect.
 - E_1 and E_2 form a successful Read-Modify-Write pair.
 - E_2 is an Explicit Memory Effect.
- All of the following apply:
 - E_1 is an Explicit Memory Effect.
 - E_1 and E_3 form a successful Read-Modify-Write pair.
 - E_2 is a Local memory read successor of E_3 .
 - One of the following applies:
 - E_2 is an Explicit Memory Read Effect with Acquire semantics.
 - E_2 is an Explicit Memory Read Effect with AcquirePC semantics.
- All of the following apply:
 - E_1 is an Implicit TTD Memory Read Effect.
 - E_1 and E_2 form a successful Read-Modify-Write pair.
 - E_2 is a Hardware Update Effect.

Barrier-ordered-before

An Effect E_1 is *Barrier-ordered-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - One of the following applies:
 - E_1 is an Explicit Memory Effect.
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DMB FULL Effect.
 - E_3 appears in program order before E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Effect.
 - E_2 is an Implicit Tag Memory Read Effect.
 - E_2 is an MMU Fault Effect.
- All of the following apply:
 - E_1 is an Explicit Memory Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DMB FULL Effect.
 - E_3 appears in program order before E_2 .
 - E_2 is a DC CVAU Effect.
- All of the following apply:
 - E_1 is a DC CVAU Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DMB FULL Effect.
 - E_3 appears in program order before E_2 .
 - E_2 is an Explicit Memory Effect.
- All of the following apply:
 - E_1 is a DC CVAU Effect.

- E_1 appears in program order before E_3 .
- E_3 is a DMB FULL Effect.
- E_3 appears in program order before E_2 .
- E_2 is a DC CVAU Effect.
- All of the following apply:
 - One of the following applies:
 - All of the following apply:
 - E_1 is an Explicit Effect.
 - E_1 is a Memory Read Effect.
 - It is not the case that E_1 is generated by an atomic instruction whose destination register is WZR or XZR.
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DMB LD Effect.
 - E_3 appears in program order before E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Effect.
 - E_2 is an Implicit Tag Memory Read Effect.
 - E_2 is an MMU Fault Effect.
- All of the following apply:
 - E_1 is an Explicit Memory Write Effect.
 - E_1 appears in program order before E_3 .
 - E_3 is a DMB ST Effect.
 - E_3 appears in program order before E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is an MMU Fault Effect.
- All of the following apply:
 - E_1 is an Explicit Memory Write Effect with Release semantics.
 - E_3 and E_1 are generated by an atomic instruction.
 - E_3 is an Explicit Memory Read Effect with Acquire semantics.
 - E_1 appears in program order before E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Effect.
 - E_2 is an Implicit Tag Memory Read Effect.
 - E_2 is an MMU Fault Effect.
- All of the following apply:
 - E_1 is an Explicit Memory Write Effect with Release semantics.
 - E_1 appears in program order before E_2 .
 - E_2 is an Explicit Memory Read Effect with Acquire semantics.
- All of the following apply:
 - One of the following applies:
 - E_1 is an Explicit Memory Read Effect with Acquire semantics.
 - E_1 is an Explicit Memory Read Effect with AcquirePC semantics.
 - E_1 appears in program order before E_2 .
 - One of the following applies:
 - E_2 is an Explicit Memory Effect.
 - E_2 is an Implicit Tag Memory Read Effect.
 - E_2 is an MMU Fault Effect.
- All of the following apply:
 - One of the following applies:
 - E_1 is an Explicit Memory Effect.
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 appears in program order before E_2 .

— E_2 is an Explicit Memory Write Effect with Release semantics.

Locally-ordered-before

An Effect E_1 is *Locally-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Intrinsically-ordered-before E_2 .
- E_1 is ETS-ordered-before E_2 .
- E_1 is Fetch-ordered-before E_2 .
- E_1 is Same-Cache-Line-ordered-before E_2 .
- E_1 is DSB-ordered-before E_2 .
- E_1 is Instruction-fetch-barrier-ordered-before E_2 .
- E_2 is a Local memory write successor of E_1 .
- All of the following apply:
 - E_3 is a Local memory write successor of E_1 .
 - E_3 belongs to the same single-copy-atomic grouping as E_2 .
- E_1 is Dependency-ordered-before E_2 .
- E_1 is Pick-ordered-before E_2 .
- E_1 is Atomic-ordered-before E_2 .
- E_1 is Barrier-ordered-before E_2 .
- All of the following apply:
 - E_1 is Locally-ordered-before E_3 .
 - E_3 is Locally-ordered-before E_2 .

Pick-locally-ordered-before

An Effect E_1 is *Pick-locally-ordered-before* an Effect E_2 if all of the following apply:

- There is a Pick dependency from E_1 to E_3 .
- E_3 is Locally-ordered-before E_2 .
- E_2 is an Explicit Memory Write Effect.

Locally-hardware-required-ordered-before

An Effect E_1 is *Locally-hardware-required-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Locally-ordered-before E_2 .
- E_1 is Pick-locally-ordered-before E_2 .
- All of the following apply:
 - E_1 is Locally-hardware-required-ordered-before E_3 .
 - E_3 is Locally-hardware-required-ordered-before E_2 .

Hardware-required-ordered-before

An Effect E_1 is *Hardware-required-ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Locally-hardware-required-ordered-before E_2 .
- E_1 is Hazard-ordered-before E_2 .

B2.4.6 Observation relations

Explicit-Observed-by

An Effect E_1 is *Explicit-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_1 is an Explicit Memory Effect.
 - E_2 Reads-from-memory E_1 .
 - E_1 and E_2 are from different Processing Elements.
 - E_2 is an Explicit Memory Effect.

- All of the following apply:
 - E_1 is an Explicit Memory Effect.
 - E_1 is Coherence-before E_2 .
 - E_1 and E_2 are from different Processing Elements.
 - E_2 is an Explicit Memory Effect.

Tag-Observed-by

An Effect E_1 is *Tag-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_1 is an Explicit Memory Write Effect.
 - E_2 Reads-from-memory E_1 .
 - E_1 and E_2 are from different Processing Elements.
 - E_2 is an Implicit Tag Memory Read Effect.
- All of the following apply:
 - E_1 is an Implicit Tag Memory Read Effect.
 - E_1 is Coherence-before E_2 .
 - E_1 and E_2 are from different Processing Elements.
 - E_2 is an Explicit Memory Write Effect.

TLBUncacheable-coherence-before

An Effect E_1 is *TLBUncacheable-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is an Implicit TTD Memory Read Effect.
- E_1 is Translation-intrinsically-before E_3 .
- E_3 is a TLBUncacheable Fault Effect.
- E_1 is Coherence-before E_2 .
- One of the following applies:
 - E_2 is an Explicit Memory Write Effect.
 - E_2 is a Hardware Update Effect.

Hardware-Update-coherence-before

An Effect E_1 is *Hardware-Update-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is an Explicit Memory Read Effect.
- E_1 is Coherence-before E_2 .
- E_2 is a Hardware Update Effect.

TLBI-coherence-before

An Effect E_1 is *TLBI-coherence-before* an Effect E_2 if all of the following apply:

- E_1 is a TLBI Effect.
- E_1 is TLBI-before E_3 .
- E_3 is an Implicit TTD Memory Read Effect.
- E_3 is Coherence-before E_2 .
- E_2 is a Memory Write Effect.

TTD-Observed-by

An Effect E_1 is *TTD-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_1 is an Implicit TTD Memory Effect.
 - E_2 Reads-from-memory E_1 .
- All of the following apply:
 - E_2 Reads-from-memory E_1 .
 - E_2 is an Implicit TTD Memory Effect.

- E_1 is TLBUncacheable-coherence-before E_2 .
- E_1 is Hardware-Update-coherence-before E_2 .
- All of the following apply:
 - E_1 is a Hardware Update Effect.
 - E_1 is Coherence-before E_2 .
 - E_2 is a Memory Write Effect.
- All of the following apply:
 - E_1 is a Memory Write Effect.
 - E_1 is Coherence-before E_2 .
 - E_2 is a Hardware Update Effect.
- E_1 is TLBI-coherence-before E_2 .

IC-Coherence-before

An Effect E_1 is *IC-coherence-before* an Effect E_2 if one of the following applies:

- All of the following apply:
 - The **Effective value** of CTR_EL0.DIC is 0.
 - The **Effective value** of CTR_EL0.IDC is 0.
 - E_1 is an IC Effect.
 - E_1 is IC-before E_3 .
 - E_3 is an Implicit Instruction Memory Read Effect.
 - E_3 is Coherence-before E_4 .
 - E_4 is a Memory Write Effect.
 - E_4 is DC-before E_2 .
 - E_2 is a DC CVAU Effect.
- All of the following apply:
 - The **Effective value** of CTR_EL0.DIC is 0.
 - The **Effective value** of CTR_EL0.IDC is 1.
 - E_1 is an IC Effect.
 - E_1 is IC-before E_3 .
 - E_3 is an Implicit Instruction Memory Read Effect.
 - E_3 is Coherence-before E_2 .
 - E_2 is a Memory Write Effect.
- All of the following apply:
 - The **Effective value** of CTR_EL0.DIC is 1.
 - The **Effective value** of CTR_EL0.IDC is 1.
 - E_1 is an Implicit Instruction Memory Read Effect.
 - E_1 is Coherence-before E_2 .
 - E_2 is a Memory Write Effect.

Instruction-Observed-by

An Effect E_1 is *Instruction-Observed-by* an Effect E_2 if one of the following applies:

- All of the following apply:
 - E_2 Reads-from-memory E_1 .
 - E_2 is an Implicit Instruction Memory Read Effect.
- E_1 is IC-before E_2 .
- All of the following apply:
 - E_1 is a DC CVAU Effect.
 - E_1 is DC-before E_2 .
 - E_2 is a Memory Write Effect.
- All of the following apply:
 - E_1 is a Memory Write Effect.
 - E_1 is DC-before E_2 .
 - E_2 is a DC CVAU Effect.

- E_1 is IC-coherence-before E_2 .

Observed-by

An Effect E_1 is *Observed-by* an Effect E_2 if one of the following applies:

- E_1 is Explicit-Observed-by either E_2 or an Effect which belongs to the same single-copy-atomic grouping as E_2 .
- E_1 is Tag-Observed-by either E_2 or an Effect which belongs to the same single-copy-atomic grouping as E_2 .
- E_1 is TTD-Observed-by E_2 .
- E_1 is Instruction-Observed-by E_2 .

B2.4.7 Definition of an Architecturally Allowed Execution

An Architecturally Allowed Execution must satisfy the *External visibility requirement*, which is formulated in terms of an ordering relation called *Ordered-before*. An execution that does not satisfy the external visibility requirement is an Architecturally Forbidden Execution.

Ordered-before

An Effect E_1 is *Ordered-before* an Effect E_2 if one of the following applies:

- E_1 is Hardware-required-ordered-before E_2 .
- E_1 is Observed-by E_2 .
- All of the following apply:
 - E_1 is Ordered-before E_3 .
 - E_3 is Ordered-before E_2 .

External visibility requirement

The external visibility requirement is defined as follows: an Architecturally Allowed Execution must not exhibit a cycle in the Ordered-before relation.

In particular, for an Effect E_1 that is Ordered-before an Effect E_2 , the External visibility requirement requires that E_2 is not Observed-by E_1 .

B2.4.8 Alternative formulations of the External visibility requirement for Explicit Memory Effects

B2.4.8.1 Additional ordering relations

Single-copy-atomic-ordered-before

A Memory Read Effect E_1 is *single-copy-atomic-ordered-before* another Memory Read Effect E_2 if all the following apply:

- E_1 belongs to the same single-copy-atomic grouping as E_2 .
- E_1 Reads-from-memory a Memory Write Effect E_3 .
- E_1 and E_3 are from different Processing Elements.
- E_2 Reads-from-memory a Memory Write Effect E_4 .
- E_2 and E_4 are from the same Processing Element.

B2.4.8.2 External completion requirement for Explicit Memory Effects

The *External completion requirement for Explicit Memory Effects* is an alternative way of formulating the External visibility requirement, in the context of Explicit Memory Effects only.

The Completes-before order is a total order that corresponds to the order in which Memory Effects complete within the system. Effects E_1 and E_2 constitute a single entry in the Completes-before order, if any of the following applies:

- All of the following apply:
 - E_1 is a Memory Write Effect.

- E_2 is a Memory Write Effect.
- E_1 belongs to the same single-copy-atomic grouping as E_2 .
- All of the following apply:
 - E_1 is a Memory Read Effect.
 - E_2 is a Memory Read Effect.
 - E_1 belongs to the same single-copy-atomic grouping as E_2 .
 - E_1 Reads-from-memory a Memory Write Effect E_3 .
 - E_1 and E_3 are from different Processing Elements.
 - E_2 Reads-from-memory a Memory Write Effect E_4 .
 - E_2 and E_4 are from different Processing Elements.
- All of the following apply:
 - E_1 is a Memory Read Effect.
 - E_2 is a Memory Read Effect.
 - E_1 belongs to the same single-copy-atomic grouping as E_2 .
 - E_1 Reads-from-memory a Memory Write Effect E_3 .
 - E_1 and E_3 are from the same Processing Element.
 - E_2 Reads-from-memory a Memory Write Effect E_4 .
 - E_2 and E_4 are from the same Processing Element.
 - E_3 belongs to the same single-copy-atomic grouping as E_4 .

All other Memory Read Effects constitute distinct entries in the Completes-before order.

Completes-before

A Memory Read or Write Effect E_1 completes-before a Memory Read or Write Effect E_2 if E_1 appears in the *Completes-before* order before E_2 .

Deriving Reads-from-memory and Coherence order from the Completes-before order

The Reads-from-memory relation can be derived from the Completes-before order by specifying which Memory Write Effect every Memory Read Effect gets its value from. Specifically, for a Memory Read Effect E_1 one of the following must apply:

- If all of the following apply:
 - There is a Memory Write Effect E_2 .
 - E_1 is a Local memory read successor of E_2 .
 - E_1 Completes-before E_2 .
 - There is no Memory Write Effect E_3 to the Same Location in the Completes-before order between E_2 and an Explicit Memory Read Effect E_4 to the Same Location that appears in program order before E_1 .
then it must be that E_1 Reads-from-memory E_2 .
- If all of the following apply:
 - There is a Memory Write Effect E_2 (possibly representing the writing of the initial value to the Location).
 - E_2 and E_1 are to the Same Location.
 - E_2 Completes-before E_1 .
 - It is not the case that E_2 Completes-before a Memory Write Effect E_3 and E_1 is the Local memory read successor of E_3 .
 - There is no Memory Write Effect E_4 to the Same Location in the Completes-before order between E_2 and E_1 .
 - There is no Memory Write Effect E_5 to the Same Location in the Completes-before order between E_2 and a Memory Read Effect E_6 to the Same Location that appears in program order before E_1 .
then it must be that E_1 Reads-from-memory E_2 .

The Coherence order can be derived from the Completes-before order by letting the Coherence order of Memory Write Effects to a memory location be the order in which those Memory Write Effects appear in the Completes-before order. The final value of each memory location is therefore determined by the

final Memory Write Effect to each Location in the Completes-before order. If no such Memory Write Effect exists for a given Location, the final value is the initial value of that Location.

External completion requirement for Explicit Memory Effects

The External completion requirement for Explicit Memory Effects requires that an Memory Effect E_1 Completes-before a Memory Effect E_2 if any of the following statements are true:

- E_1 is Locally-hardware-required-ordered-before E_2 .
- E_1 is a Memory Read Effect, E_2 is a Memory Read Effect, and E_1 is single-copy-atomic-ordered-before E_2 .

B2.4.8.3 External global completion requirement for Explicit Memory Effects

The *External global completion requirement for Explicit Memory Effects* is an alternative way of formulating the External visibility requirement, in the context of Explicit Memory Effects only.

The Globally-completes-before order is a total order that corresponds to the order in which Memory Effects complete within the system. Effects E_1 and E_2 constitute a single entry in the Globally-completes-before order, if any of the following applies:

- All of the following apply:
 - E_1 is a Memory Write Effect.
 - E_2 is a Memory Write Effect.
 - E_1 belongs to the same single-copy-atomic grouping as E_2 .
- All of the following apply:
 - E_1 is a Memory Read Effect.
 - E_2 is a Memory Read Effect.
 - E_1 belongs to the same single-copy-atomic grouping as E_2 .
 - E_1 Reads-from-memory a Memory Write Effect E_3 .
 - E_1 and E_3 are from different Processing Elements.
 - E_2 Reads-from-memory a Memory Write Effect E_4 .
 - E_2 and E_4 are from different Processing Elements.
- All of the following apply:
 - E_1 is a Memory Read Effect.
 - E_2 is a Memory Read Effect.
 - E_1 belongs to the same single-copy-atomic grouping as E_2 .
 - E_1 Reads-from-memory a Memory Write Effect E_3 .
 - E_1 and E_3 are from the same Processing Element.
 - E_2 Reads-from-memory a Memory Write Effect E_4 .
 - E_2 and E_4 are from the same Processing Element.
 - E_3 belongs to the same single-copy-atomic grouping as E_4 .

All other Memory Read Effects constitute distinct entries in the Globally-completes-before order.

Globally-completes-before

A Memory Read or Write Effect E_1 *Globally-completes-before* a Memory Read or Write Effect E_2 if E_1 appears in the Globally-completes-before order before E_2 .

Deriving Reads-from-memory and Coherence order from the Globally-completes-before order

The Reads-from-memory relation can be derived from the Globally-completes-before order by specifying which Memory Write Effect every Memory Read Effect gets its value from. Specifically, for a Memory Read Effect E_1 it must be that if all of the following applies:

- There is Memory Write Effect E_2 (possibly representing the writing of the initial value to the Location).
- E_2 Globally-completes-before E_1 .
- E_1 and E_2 are to the Same Location.

- There is no Memory Write Effect E_3 to the Same Location in the Globally-completes-before order between E_2 and E_1 .
- There is no Memory Write Effect E_4 to the Same Location in the Globally-completes-before order between E_2 and an Explicit Memory Read Effect E_5 to the Same Location that appears in program order before E_1 .

then it must be that E_1 Reads-from-memory E_2 .

The Coherence order can be derived from the Globally-completes-before order by letting the Coherence order of Memory Write Effects to a memory location be the order in which those Memory Write Effects appear in the Globally-completes-before order. The final value of each memory location is therefore determined by the final Memory Write Effect to each Location in the Globally-completes-before order. If no such Memory Write Effect exists for a given Location, the final value is the initial value of that Location.

External global completion requirement

The External global completion requirement requires that a Memory Effect E_1 Globally-completes-before a Memory Effect E_2 if any of the following statements are true:

- E_1 is Locally-hardware-required-ordered-before E_2 and one of the following applies:
 - E_1 is a Memory Write Effect.
 - E_1 is a Memory Read Effect, and one of the following applies:
 - E_1 Reads-from-memory a Memory Write Effect E_3 , and E_1 and E_3 are from different Processing Elements, or.
 - E_1 Reads-from-memory a Memory Write Effect E_3 , E_1 , and E_3 are from the same Processing Element, and E_3 is Locally-hardware-required-ordered-before E_2 .
- E_1 is a Memory Read Effect, E_2 is a Memory Read Effect, and E_1 is single-copy-atomic-ordered before E_2 .

B2.5 Additional ordering requirements outside of the scope of the formal concurrency model

The following sections present ordering rules which have been architected but not yet formalized. As a consequence they do not appear in the [formal concurrency model](#), nor in the corresponding transliteration in [Ordering requirements defined by the formal concurrency model](#). They will be integrated into [Ordering requirements defined by the formal concurrency model](#) in due course.

B2.5.1 Completion and endpoint ordering

For all memory, the completion rules are defined as:

- A Memory Read effect E_1 to a [Location](#) is complete for a shareability domain when all of the following are true:
 - Any Memory Write Effect to the same [Location](#) by an observer within the shareability domain will be Coherence-after E_1 .
 - All Implicit TTD Memory Read Effects associated with E_1 are complete for that shareability domain.
- A Memory Write effect E_1 to a [Location](#) is complete for a shareability domain when all of the following are true:
 - Any Memory Write Effect to the same [Location](#) by an observer within the shareability domain will be Coherence-after E_1 .
 - Any Memory Read Effect to the same [Location](#) by an observer within the shareability domain will be such that either [Reads-from](#) E_1 or [Reads-from](#) a Memory Write effect that is Coherence-after E_1 .
 - All Implicit TTD Memory Read Effects associated with E_1 are complete for that shareability domain.
- An Implicit TTD Memory Read Effect is complete for a shareability domain when the memory accesses associated with the Implicit TTD Memory Read Effect, including Hardware Update Effects, are complete for that shareability domain, and the TLB is updated.
- A cache maintenance instruction is complete for a shareability domain when the memory effects of the instruction are complete for that shareability domain, and Implicit TTD Memory Read Effects that arise from the instruction are complete for that shareability domain.
- A TLB invalidate instruction is complete when all memory accesses using the TLB entries that have been invalidated are complete.

The completion of any cache or TLB maintenance instruction includes its completion on all PEs that are affected by both the instruction and the DSB operation that is required to guarantee visibility of the maintenance instruction.

Additionally, for Device-nGnRnE memory, a read or write of a [Location](#) in a [Memory-mapped peripheral](#) that exhibits side effects is complete only when the read or write both:

- Can begin to affect the state of the [Memory-mapped peripheral](#).
- Can trigger all associated side effects, whether they affect other peripheral devices, PEs, or memory.

Interaction between observers in a system is not restricted to communication through shared variables in coherent memory. For example, an observer could configure an interrupt controller to raise an interrupt on another observer as a form of message passing. These interactions typically involve an additional agent, which defines the instruction sequence that is required to establish communication links between different observers. When these forms of interaction are used in conjunction with shared variables, a DSB instruction can be used to enforce ordering between them.

Note

This requirement for Device-nGnRnE memory is consistent with the memory access having reached the peripheral endpoint.

Note

These completion rules mean that, for example, a cache maintenance instruction that operates by VA to the PoC completes only after memory at the PoC has been updated.

B2.5.1.1 Peripherals

This section defines a Memory-mapped peripheral and the total order of reads and writes to a peripheral which is defined as the [Peripheral arrival order](#):

Memory-mapped peripheral

A *Memory-mapped peripheral* occupies a memory region of IMPLEMENTATION DEFINED size and can be accessed using load and store instructions. [Memory effects](#) to a Memory-mapped peripheral can have side effects, such as causing the peripheral to perform an action. Values that are read from addresses within a Memory-mapped peripheral might not correspond to the last data value written to those addresses. As such, [Memory effects](#) to a Memory-mapped peripheral might not appear in the [Reads-from](#) or [Coherence order](#) relations.

Peripheral arrival order

The *Peripheral arrival order* of a [Memory-mapped peripheral](#) is a total order on all Memory Read Effects and Memory Write Effects to that peripheral.

A [Memory effect](#) E_1 is inserted in the Peripheral arrival order for a peripheral before a [Memory effect](#) E_2 , if E_1 and E_2 are to the same peripheral and any of the following applies:

- All of the following apply:
 - E_1 is to Device memory with the non-Reordering attribute.
 - E_2 is to Device memory with the non-Reordering attribute.
 - E_1 appears in program order before E_2 .
- All of the following apply:
 - E_1 is to Device memory with the non-Gathering attribute.
 - E_2 is to Device memory with the non-Gathering attribute.
 - E_1 is Ordered-before E_2 .
- All of the following apply:
 - Any of the following applies:
 - E_1 is to Device memory.
 - E_1 is to Non-cacheable Normal memory.
 - Any of the following applies:
 - E_2 is to Device memory.
 - E_2 is to Non-cacheable Normal memory.
 - E_1 is Barrier-ordered-before E_2 .

Otherwise, exactly one of the following applies:

- E_1 and E_2 are gathered into a single transaction and appear as one Effect in the Peripheral arrival order for the peripheral.
- E_1 is inserted in the Peripheral arrival order for the peripheral before E_2 .
- E_2 is inserted in the Peripheral arrival order for the peripheral before E_1 .

Gathering is permitted, but not required if none of the following apply:

- E_1 is to Device memory with the non-Gathering attribute.
- All of the following apply:
 - E_1 appears in program order before E_2 .
 - E_2 is a [Memory Write Effect](#).
- E_2 is to Device memory with the non-Gathering attribute.

Out-of-band-ordered-before

An Effect E_1 is *Out-of-band-ordered-before* an Effect E_2 if and only if all of the following apply:

- E_1 is a Memory Effect.
- Any of the following applies:
 - E_1 is DSB-ordered-before E_3 .
 - E_1 is Instruction-fetch-barrier-ordered-before E_3 .
- There is an IMPLEMENTATION DEFINED instruction sequence beginning with E_3 indirectly leading to the generation of E_2 .
- E_2 is a Memory Effect.

If an Effect E_1 is Out-of-band-ordered-before an Effect E_2 , then E_1 is Ordered-before E_2 .

Note

Arm expects that, in most systems with early acknowledgments, those acknowledgments will come from a point at or after the point that establishes global visibility. This is expected in such systems to enable the acknowledgments to be used as part of the mechanisms to implement the ordering requirements of the Arm memory model.

B2.5.2 Metadata Dependencies**I_{YRHC}**

The execution of the following instructions create ordering relations for the purposes of the Arm memory model:

- PACDA.
- PACDB.
- AUTDA.
- AUTDB

R_{MSWTM}

If FEAT_PAAuth is implemented, the execution of a PACDA Xd, Xn instruction creates:

- A [Dependency through registers and memory](#) from the Register Read Effect of Xd to Register Write Effect of Xd.
- If authentication using the APDAKey_EL1 key is enabled, a [Dependency through registers and memory](#) from the Register Read Effect of Xn to Register Write Effect of Xd.

R_{JMNSC}

If FEAT_FPAC is implemented, the execution of a PACDB Xd, Xn instruction creates:

- A [Dependency through registers and memory](#) from the Register Read Effect of Xd to Register Write Effect of Xd.
- If authentication using the APDBKey_EL1 key is enabled, a [Dependency through registers and memory](#) from the Register Read Effect of Xn to Register Write Effect of Xd.

R_{XYRJT}

If FEAT_FPAC is implemented, the execution of an AUTDA Xd, Xn instruction creates:

- If authentication using the APDAKey_EL1 key is enabled, a [Pick dependency through registers and memory](#) from the Register Read Effect of Xn to Register Write Effect of Xd.
- A [Dependency through registers and memory](#) from the Register Read Effect of Xd to Register Write Effect of Xd.

R_{SPBZX}

If FEAT_FPAC is implemented, the execution of an AUTDB Xd, Xn instruction creates:

- If authentication using the APDBKey_EL1 key is enabled, a [Pick dependency through registers and memory](#) from the Register Read Effect of Xn to Register Write Effect of Xd.
- A [Dependency through registers and memory](#) from the Register Read Effect of Xd to Register Write Effect of Xd.

B2.5.3 Limited ordering regions

FEAT_LOR introduces *limited ordering regions* (LORegions), which allow large systems to perform special load-acquire and store-release instructions that provide order between the memory accesses to a region of the PA map as observed by a set of observers.

LORegions are defined in the Non-secure physical address space. LORegions cannot be defined in the Secure, Realm, or Root physical address spaces.

This feature is supported in AArch64 state only.

B2.5.4 Ordering of instruction fetches

For two memory locations A and B, if A has been written to with an updated value and been made coherent with the instruction fetches of the shareability domain before B has been written to with an updated value by an observer in the same shareability domain, then where, for an observer in the shareability domain, an instruction read from B appears in program order before an instruction fetched from A, if the instruction read from B contains the updated value of B then the instruction read from A appearing later in program order will contain the updated value of A.

A write has been made coherent with an instruction fetch of a shareability domain when:

CTR.{DIC, IDC} = {0, 0} The location written to has been cleaned to the [Point of unification \(PoU\)](#) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the [Point of unification \(PoU\)](#) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} = {0, 1} The write is complete for the shareability domain. Subsequently the location has been invalidated to the [Point of unification \(PoU\)](#) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} = {1, 1} The write is complete for the shareability domain.

Note

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
 - For an implementation:
 - When **CTR.DIC** = 0, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the [ICIMVAU](#) instructions of a different core.
 - When **CTR.DIC** = 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.
-

B2.5.5 Memory barriers

The Arm architecture is a weakly ordered memory architecture that supports out of order completion. *Memory barrier* is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring load/store instructions. The memory barriers defined by the architecture provide a range of functionality, including:

- Ordering of load/store instructions.
- Completion of load/store instructions.
- Context synchronization.

The Armv8-R AArch32 architecture redefines **DMB** and **DSB**, and defines a new instruction, **DFB**.

Armv8-R relaxes the ordering requirements for **DMB** and **DSB** by enforcing ordering only in terms of certain Exception level criteria:

- Accesses from EL1 and EL0 are ordered only with respect to accesses using the same VMID.
- Accesses from EL2 are ordered only with respect to other accesses from EL2.

All other memory barrier requirements are unchanged.

The following subsections describe the memory barrier instructions:

- [Instruction Synchronization Barrier](#).
- [Data Memory Barrier](#).
- [Data Synchronization Barrier](#).
- [Load-Acquire, Store-Release](#).

Note

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are available only when software execution is at EL1 or higher.

The DMB and DSB memory barriers affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance instructions being executed by the PE.

AArch32 state also supports the legacy barrier instructions [CP15DMB](#), [CP15DSB](#), and [CP15ISB](#). These instructions are executed as MCRs using the appropriate encoding, and are accessible from EL0. However, for performance reasons Arm deprecates any use of these operations, and strongly recommends that software uses the [DMB](#), [DSB](#), and [ISB](#) instructions described in this section instead. Optionally, an implementation can support a CP15BEN control that supervisory software can use to disable use of these instructions, meaning the corresponding MCR encodings are UNDEFINED. When the CP15BEN control is supported, setting one of the following CP15BEN fields to 0 makes execution of [CP15DMB](#), [CP15DSB](#), and [CP15ISB](#) UNDEFINED:

- [SCTLR](#).CP15BEN, for execution of these instructions at EL0 or EL1 when EL1 is using AArch32.
- [HSTCLR](#).CP15BEN, for execution of these instructions at EL2 when EL2 is using AArch32.

B2.5.5.1 Instruction Synchronization Barrier

An [ISB](#) instruction ensures that all instructions that come after the [ISB](#) instruction in program order are fetched from the cache or memory after the [ISB](#) instruction has completed. Using an [ISB](#) ensures that the effects of context-changing operations executed before the [ISB](#) are visible to the instructions fetched after the [ISB](#) instruction. Examples of context-changing operations that require the insertion of an [ISB](#) instruction to ensure the effects of the operation are visible to instructions fetched after the [ISB](#) instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the [ISB](#) instruction take effect only after the [ISB](#) has been executed.

The pseudocode function for the operation of an [ISB](#) is [InstructionSynchronizationBarrier\(\)](#).

See also [Memory barriers](#).

B2.5.5.2 Data Memory Barrier

The [DMB](#) instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The [DMB](#) instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

The full definition of the [DMB](#) instruction is covered formally in the [Definition of the memory model](#) and this introduction to the [DMB](#) instruction is not intended to contradict that section.

The basic principle of a [DMB](#) instruction is to introduce order between memory accesses that are specified to be affected by the [DMB](#) options supplied as arguments to the [DMB](#) instruction. The [DMB](#) instruction ensures that all affected memory accesses by the PE executing the [DMB](#) instruction that appear in program order before the [DMB](#) instruction and those which originate from a different PE, to the extent required by the [DMB](#) options, which have been Observed-by the PE before the [DMB](#) instruction is executed, are Observed-by each PE, to the extent required by the [DMB](#) options, before any affected memory accesses that appear in program order after the [DMB](#) instruction are Observed-by that PE.

The use of a [DMB](#) instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The pseudocode function for the operation of a [DMB](#) instruction is [DataMemoryBarrier\(\)](#).

B2.5.5.3 Data Synchronization Barrier

A [DSB](#) instruction is a memory barrier that ensures that memory accesses that occur before the [DSB](#) instruction have completed before the completion of the [DSB](#) instruction. In doing this, it acts as a stronger barrier than a [DMB](#) and all ordering that is created by a [DMB](#) with specific options is also generated by a [DSB](#) with the same options.

Execution of a [DSB](#) at EL2 ensures that any memory accesses caused by speculative accesses from the PL1&0 translation regime have been observed.

A [DSB](#) executed by a PE, PEe, completes when all of the following apply:

- All explicit memory effects of the required access types appearing in program order before the [DSB](#) are complete for the set of observers in the required shareability domain.

- If the required access types of the DSB is reads and writes, the following instructions issued by PEE before the DSB are complete for the required shareability domain:
 - All cache maintenance instructions.
 - All PSB instructions.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes, other than:

- Being fetched from memory and decoded, or any change to architectural or microarchitectural state resulting from being fetched or decoded.
- Indirectly or directly reading any of the following, so long as the reading of these resources does not cause a change in architectural state:
 - General-purpose registers
 - SIMD&FP registers
 - The Stack Pointer register
 - The Program Counter
 - Special-purpose registers

The pseudocode function for the operation of a DSB is `DataSynchronizationBarrier()`.

See also [Memory barrier instructions](#) and [Memory barriers](#).

B2.5.5.3.1 Maintenance scope of the data synchronization barrier operations

A DSB instruction that applies to both reads and writes takes an argument that specifies its maintenance scope. The maintenance scope indicates the subset of TLB invalidate (TLBI) instructions, TLB invalidate pair (TLBIP) instructions, and Instruction cache maintenance (IC) instructions that are covered by this DSB, for one of:

- Outer Shareable domain.
- Inner Shareable domain.
- Non-shareable domain.

When a DSB instruction that applies to both reads and writes is used to ensure completion of TLB maintenance instructions or Instruction cache maintenance (IC) instructions, all of the following apply:

- A DSB_{NSH} is sufficient to ensure completion of TLB maintenance instructions or Instruction cache maintenance (IC) instructions that apply to a single PE.
- A DSB_{ISH} is sufficient to ensure completion of TLB maintenance instructions or Instruction cache maintenance (IC) instructions that apply to PEs in the same Inner Shareable domain.
- A DSB_{OSH} is sufficient to ensure completion of TLB maintenance instructions that apply to PEs in the same Outer Shareable domain.

B2.5.5.4 Load-Acquire, Store-Release

The architecture provides a set of instructions with Acquire semantics for loads, and Release semantics for stores.

The full definition of the Load-Acquire instruction is covered formally in the [Definition of the memory model](#) and this introduction to the Load-Acquire instruction is not intended to contradict that section.

The basic principle of a Load-Acquire instruction is to introduce order between the memory access generated by the Load-Acquire instruction and the memory accesses appearing in program order after the Load-Acquire instruction, such that the memory access generated by the Load-Acquire instruction is Observed-by each PE, to the extent that PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire instruction are Observed-by that PE, to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the [Definition of the memory model](#) and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the memory accesses generated by the PEE executing the Store-Release instruction, together with those which originate from a different PE, to the extent that the PEE is required to observe them coherently, Observed-by the PEE before executing the Store-release.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

In addition, the use of a Load-Acquire or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral arrival order.

Load-Acquire and Store-Release, other than LDAEXD and STLEXD, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.

LDAEXD and STLEXD access two data elements. The address supplied to the instructions must be doubleword-aligned, otherwise the access generates an Alignment fault.

A Store-Release Exclusive instruction has the release semantics only if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
 - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
 - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
 - The Load-Acquire/Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction.
-

Table B2-1 summarizes the Load-Acquire/Store-release instructions.

Table B2-1 Load-Acquire/Store-Release instructions

Data type	Load-Acquire	Store-Release	Load-Acquire Exclusive	Store-Release Exclusive
32-bit word	LDA	STL	LDAEX	STLEX
16-bit halfword	LDAH	STLH	LDAEXH	STLEXH
8-bit byte	LDAB	STLB	LDAEXB	STLEXB
64-bit doubleword	-	-	LDAEXD	STLEXD

B2.6 Endian support

[General description of endianness in the Arm architecture](#) describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- [Instruction endianness.](#)
- [Data endianness.](#)
- [Endianness of memory-mapped peripherals.](#)

B2.6.1 General description of endianness in the Arm architecture

This section only describes memory addressing and the effects of endianness for data elements up to doubleword of 64 bits. However, this description can be extended to apply to larger data elements.

For an address A, [Figure B2-1](#) shows, for big-endian and little-endian memory systems, the relationship between:

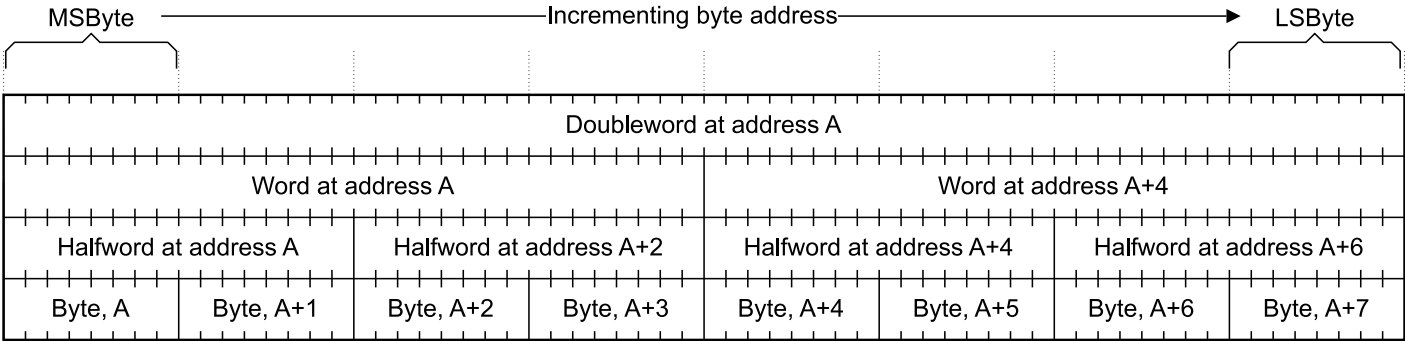
- The doubleword at address A.
- The words at addresses A and A+4.
- The halfwords at addresses A, A+2, A+4, and A+6.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, and A+7.

The terms in [Figure B2-1](#) have the following definitions:

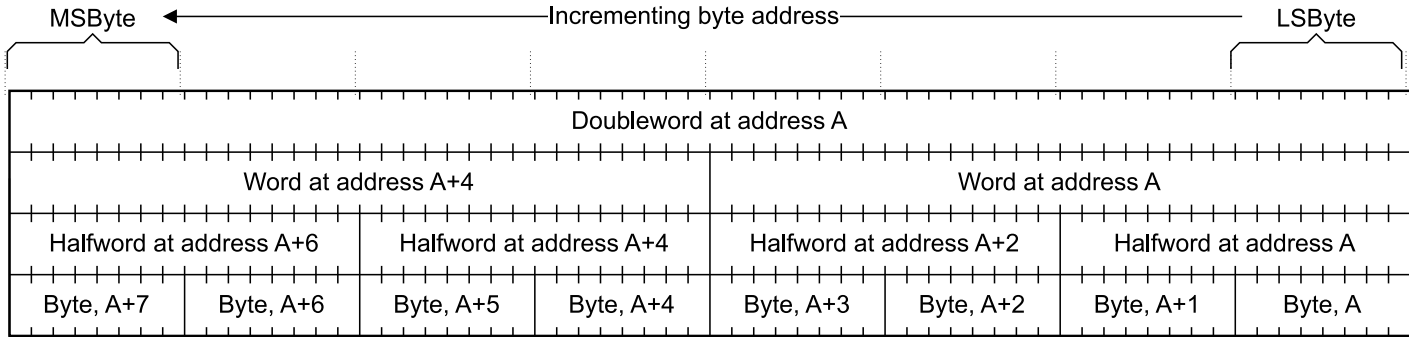
MSByte Most significant byte.

LSByte Least significant byte.

Big-endian memory system



Little-endian memory system



In this figure, *Byte, A+1* is an abbreviation for *Byte at address A+1*

Figure B2-1 Endianness relationships in AArch32 state

B2.6.2 Instruction endianness

The mapping of instruction memory is always little-endian.

B2.6.3 Data endianness

The size of the data value that is loaded or stored is the size that is used for the purpose of endian conversion for floating-point, Advanced SIMD, and general-purpose register loads and stores.

Table B2-2 shows the element sizes of all the load/store instructions, for all instruction sets.

Table B2-2 Element size of load/store instructions

Instructions	Element size
LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, TBB	Byte
LDRH, LDREXH, LDRHT, LDRSH, LDRSHT, STRH, STREXH, STRHT, TBH	Halfword
LDR, LDRT, LDREX, STR, STRT, STREX	Word
LDRD, LDREXD, STRD, STREXD	Word
All forms of LDM, PUSH, POP, RFE, SRS, all forms of STM,	Word
LDC, STC	Word

Instructions	Element size
Forms of VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR that transfer 32-bit S registers	Word
Forms of VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR that transfer 64-bit D registers	Doubleword
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4	Element size of the Advanced SIMD access

[CPSR.E](#) determines the data endianness.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for Advanced SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for Advanced SIMD element and data structure loads and stores. For more information, see [Endianness in Advanced SIMD](#).

B2.6.3.1 Instructions to reverse bytes in registers

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

[Table B2-3](#) shows the instructions that provide this functionality in the T32 and A32 instruction sets.

Table B2-3 Byte reversal instructions

Function	T32/A32 instruction	Notes
Reverse bytes in whole register	REV	For use with general purpose registers
Reverse bytes in 16-bit halfwords	REV16	For use with general purpose registers
Reverse bytes in halfword and sign-extend	REVSH	For use with general purpose registers
Reverse elements in doublewords, vector	VREV64	For use with registers in the SIMD and floating-point register file
Reverse elements in words, vector	VREV32	For use with registers in the SIMD and floating-point register file
Reverse elements in halfwords, vector	VREV16	For use with registers in the SIMD and floating-point register file

B2.6.3.2 Endianness in Advanced SIMD

Advanced SIMD element load/store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used by the PE to load and store data correctly in both big-endian and little-endian systems.

Consider, for example, the A32 or T32 instruction:

```
VLD1.16 {D0}, [R1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in [Figure B2-2](#). Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

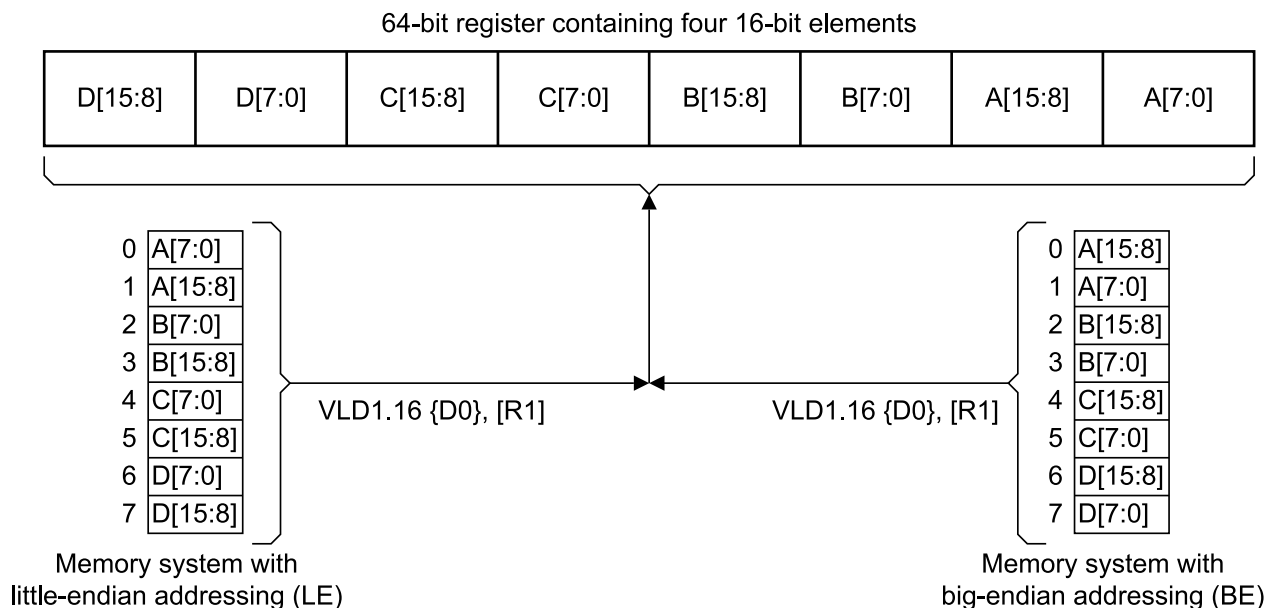


Figure B2-2 Advanced SIMD byte order example for AArch32 state

For information about the alignment of Advanced SIMD instructions, see [Alignment support](#).

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in [A-profile Architecture Pseudocode](#).

B2.6.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a Cross Trigger Interface, see [About the External Debug Registers](#).
- The memory-mapped register interface to the system level implementation of the Generic Timer, see [System Level Implementation of the Generic Timer](#).
- A memory-mapped register interface to the Performance Monitors, see [Recommended External Interface to the Performance Monitors](#).
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM[®] Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4*.
- The memory-mapped register interface to an Arm trace component. See, for example, the *Arm[®] Embedded Trace Macrocell Architecture Specification, ETMv4*.

B2.7 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. The architecture defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- [Introduction to caches.](#)
- [Memory hierarchy.](#)
- [Implication of caches for the application programmer.](#)
- [Prefetching into cache.](#)

B2.7.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access and take account of two principles of locality:

Spatial locality

An access to one location is likely to be followed by accesses to adjacent locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. The architecture permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

B2.7.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, a memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. [Figure B2-3](#) shows an example of such a system in a system that supports virtual addressing.

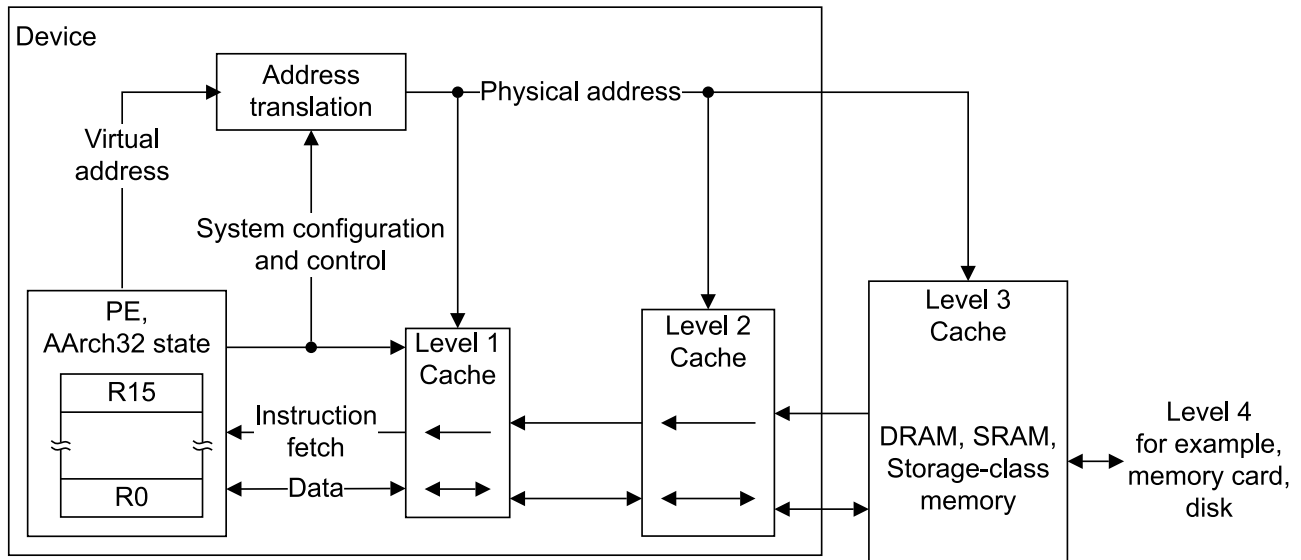


Figure B2-3 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the PE, as shown in [Figure B2-3](#).

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU) and Point of Coherency (PoC). For more information, including the definitions of PoU and PoC, see [About cache maintenance in AArch32 state](#).

B2.7.2.1 The Cacheability and Shareability memory attributes

Cacheability and Shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

- Cacheability** This term defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.
- Shareability** This term defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about the Cacheability and Shareability attributes, see [Memory types and attributes](#).

B2.7.3 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.

- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

B2.7.3.1 Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
 - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
 - Not enabling caches in the system.
- By using system calls to functions using cache maintenance instructions that execute at a higher Exception level.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see [Non-shareable Normal memory](#) and [Shareable, Inner Shareable, and Outer Shareable Normal memory](#).

Note

The performance of these hardware coherency mechanisms is highly IMPLEMENTATION SPECIFIC. In some implementations the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

B2.7.3.2 Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last [Context Synchronization event](#) on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being re-fetched from memory.

The Arm architecture does not require the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory.

If software requires coherency between instruction execution and memory, it must manage this coherency using [Context Synchronization events](#) and cache maintenance instructions. These can be accessed only from an Exception level that is higher than EL0, and therefore require a system call, see [Exception-generating and exception-handling instructions](#). The following code sequence can be used for this purpose:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.
STR Rt, [Rn]
DCCMVAU Rn      ; Clean data cache by MVA to point of unification (PoU)
DSB             ; Ensure visibility of the data cleaned from cache
ICIMVAU Rn      ; Invalidate instruction cache by MVA to PoU
BPIMVA Rn       ; Invalidate branch predictor by MVA to PoU
DSB             ; Ensure completion of the invalidations
ISB             ; Synchronize the fetched instruction stream
```

A write has been made coherent with an instruction fetch of a shareability domain when:

CTR.{DIC, IDC} = {0, 0} The location written to has been cleaned to the [Point of unification \(PoU\)](#) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the [Point of unification \(PoU\)](#) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} = {0, 1} The write is complete for the shareability domain. Subsequently the location has been invalidated to the [Point of unification \(PoU\)](#) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR.{DIC, IDC} = {1, 1} The write is complete for the shareability domain.

Note

- For accesses that are Non-cacheable or Write-Through, the clean data cache instruction is not required. For accesses that are Non-cacheable, the invalidate instruction cache is not required, because in AArch32 state these accesses are not permitted to be held in an instruction cache.
 - This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See [Concurrent modification and execution of instructions](#).
-

B2.7.4 Prefetching into cache

The Arm architecture provides the memory system hints `PLD` (Preload Data), `PLDW` (Preload Data With Intent To Write) and `PLI` (Preload Instruction) that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring data or instruction locations into caches.

The prefetch instructions are hints, and so implementations can treat them as `NOPS` without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is reported using an `SError` exception and taken using an asynchronous Data Abort exception. For more information, see [Data Abort exception](#).

A `PLD`, `PLDW`, or `PLI` instruction can only cause allocation to software-visible caching structures such as caches or TLBs for memory locations that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

A `PLD`, `PLDW`, or `PLI` instruction can access any memory location in Normal memory that can be accessed, according to the permissions defined by the current translation regime or a translation regime for a higher Exception level in the current Security state, by any of:

- Reads.
- Writes.
- Instruction fetches.

Note

In each case, the entire list applies to each of `PLD`, `PLDW`, and `PLI`.

A `PLD`, `PLDW`, or `PLI` instruction is guaranteed not to access any type of Device memory.

A `PLI` instruction must not perform any access that cannot be performed by a speculative instruction fetch by the processor.

The pseudocode enumeration [PrefetchHint](#) defines the prefetch hint types.

The `Hint_Prefetch()` pseudocode function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as prefetching the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

For more information on `PLD`, `PLI`, and `PLDW`, see:

- [PLD, PLDW \(immediate\)](#).

- PLD (literal).
- PLD, PLDW (register).
- PLI (immediate, literal).
- PLI (register).

B2.8 Alignment support

This section describes alignment support. It contains the following subsections:

- [Instruction alignment.](#)
- [Unaligned data access.](#)
- [Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE.](#)
- [Unaligned data access restrictions.](#)

For more information about Alignment faults, see [Alignment faults](#).

B2.8.1 Instruction alignment

A32 instructions are word-aligned.

T32 instructions are halfword-aligned.

B2.8.2 Unaligned data access

An R-profile implementation must support unaligned data accesses to Normal memory by some load and store instructions. As [Table B2-4](#) shows, software can control whether a misaligned access to Normal memory by one of these instructions causes an Alignment fault Data Abort exception:

- By setting [SCTLR.A](#), for unaligned accesses from any mode other than Hyp mode.
- By setting [HSCTLR.A](#), for unaligned accesses from Hyp mode.

Table B2-4 Alignment requirements of load/store instructions

Instructions	Alignment check	Result if check fails when:	
		SCTLR.A or HSCTLR.A is 0	SCTLR.A or HSCTLR.A is 1
LDRB, LDREXB, LDRBT, LDRSB, LDRSBT, STRB, STREXB, STRBT, TBB	None	-	-
LDRH, LDRHT, LDRSH, LDRSHT, STRH, STRHT, TBH	Halfword	Unaligned access	Alignment fault
LDREXH, STREXH, LDAH, STLH, LDAEXH, STLEXH	Halfword	Alignment fault	Alignment fault
LDR, LDRT, STR, STRT PUSH, encodings T3 and A2 only POP, encodings T3 and A2 only	Word	Unaligned access	Alignment fault
LDREX, STREX, LDA, STL, LDAEX, STLEX	Word	Alignment fault	Alignment fault
LDREXD, STREXD, LDAEXD, STLEXD	Doubleword	Alignment fault	Alignment fault
All forms of LDM and STM, LDRD, RFE, SRS, STRD	Word	Alignment fault	Alignment fault
LDC, STC	Word	Alignment fault	Alignment fault
VLDM, VPOP, VPUSH, VSTM	Word	Alignment fault	Alignment fault
VLDR, VSTR - single-precision scalar and double-precision scalar	Word	Alignment fault	Alignment fault
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with standard alignment	Element size	Unaligned access	Alignment fault
VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4, all with :<align> specified ^a	As specified by :<align>	Alignment fault	Alignment fault

a. Previous versions of this manual used @<align> to specify alignment. Both forms are supported, see [T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions](#) for more information.

Note

Any unaligned access to any type of Device memory generates an Alignment fault, see [Alignment faults](#).

B2.8.3 Cases where unaligned accesses are CONSTRAINED UNPREDICTABLE

Any load instruction that is not faulted by the alignment restrictions shown in [Table B2-4](#) and that loads the PC has CONSTRAINED UNPREDICTABLE behavior if the address it loads from is not word-aligned, see [Loads and Stores to unaligned locations](#). This overrides any permitted load/store behavior shown in [Table B2-4](#).

B2.8.4 Unaligned data access restrictions

The following points apply to unaligned data accesses:

- Accesses are not guaranteed to be single-copy atomic except at the byte access level, see [Atomicity in the Arm architecture](#).
- Unaligned accesses typically take a number of additional cycles to complete compared to a naturally-aligned access.
- An operation that performs an unaligned access can abort on any memory access that it makes, and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the boundary.

B2.8.5 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a cross-trigger interface, see [About the External Debug Registers](#).
- The memory-mapped register interface to the system level implementation of the Generic Timer, see [System Level Implementation of the Generic Timer](#).
- A memory-mapped register interface to the Performance Monitors, see [Recommended External Interface to the Performance Monitors](#).
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM[®] Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4*.
- The memory-mapped register interface to an Arm trace component. See, for example, the *Arm[®] Embedded Trace Macrocell Architecture Specification, ETMv4*.

B2.9 Memory types and attributes

The ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- [Normal memory](#).
- [Device memory](#).
- [Memory access restrictions](#).

B2.9.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform [Speculative](#) data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- An Explicit Memory Write Effect to a memory location with the Normal attribute completes in finite time.
- Explicit Memory Write Effects to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer Cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes. For the purposes of this requirement, the endpoint for a location in [Conventional memory](#) is the PoC.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See [Multi-register loads and stores that access Normal memory](#).

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
 - Read accesses can be repeated with no side-effects.
 - Repeated read accesses return the last value written to the resource being read.
 - Read accesses can fetch additional memory locations with no side-effects.
 - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
 - Unaligned accesses can be supported.
 - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
 - Memory accesses return UNKNOWN values.
 - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in [Atomicity in the Arm architecture](#) might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

The following sections describe the other attributes for Normal memory:

- [Shareable Normal memory](#).
- [Non-shareable Normal memory](#).
- [Cacheability attributes for Normal memory](#).

See also:

- [Multi-register loads and stores that access Normal memory.](#)
- [Atomicity in the Arm architecture.](#)
- [Memory barriers.](#) For accesses to Normal memory, a DMB instruction is required to ensure the required ordering.
- [Concurrent modification and execution of instructions.](#)

B2.9.1.1 Shareable Normal memory

A Normal memory location has a Shareability attribute that is defined as one of:

- Inner Shareable.
- Outer Shareable.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, which hardware must enforce. They do not affect the coherency requirements of instruction fetches, see [Synchronization and coherency issues between data and instruction accesses](#).

Note

- System designers can use the Shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the Shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different PEs.
 - This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.
-

B2.9.1.1.1 Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is only a member of a single Inner Shareability domain.
- Each observer is only a member of a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Note

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
 - The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.
-

The details of the use of the Shareability attributes are system-specific. [Example B2-1](#) shows how they might be used.

Example B2-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
 - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
 - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different Shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same Shareability domain for the Outer Shareable attribute.

A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable Shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable Shareability domain.

For Shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

B2.9.1.2 Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

B2.9.1.3 Cacheability attributes for Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Back Cacheable.
- Write-Through.
- Non-cacheable.

Also, for Write-Back Cacheable and Write-Through Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

For more information, see [Cacheability, cache allocation hints, and cache transient hints](#).

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint this does not affect the behavior of accesses to that location.
- For other cases, see [Mismatched memory attributes](#).

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the Shareability domain of a region of memory. In some cases, the use of Write-Through or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. For this reason, the architecture requires the following properties for Non-cacheable or Write-Through memory:

- A completed write to a memory location that is Non-cacheable or Write-Through for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable or Write-Through for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.
- For accesses to Normal memory that is Non-cacheable or Write-Through, a DMB instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations and constraints.

Note

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner Cacheability attributes can lie outside a cache that is controlled by the Outer Cacheability attributes.
- An implementation might not have any outer cache.

[Example B2-2](#), [Example B2-3](#), and [Example B2-4](#) describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

Note

- L1 cache is the level closest to the PE, see [Memory hierarchy](#).
 - When managing coherency, system designs must consider both the Inner and Outer Cacheability attributes, as well as the Shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.
-

Example B2-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner Cacheability attribute applied to L1 and L2 cache.
- The Outer Cacheability attribute applied to L3 cache.

Example B2-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner Cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer Cacheability attribute.

Example B2-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner Cacheability attribute applied to L1 cache.
- The Outer Cacheability attribute applied to L2 and L3 cache.

B2.9.1.4 Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more registers from the SIMD and floating-point register file from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

B2.9.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Device memory are:

Gathering Identified as G or nG, see [Gathering](#).

Reordering Identified as R or nR, see [Reordering](#).

Early Write Acknowledgement

Identified as E or nE, see [Early Write Acknowledgement](#).

The Device memory types are:

Device-nGnRnE Device non-Gathering, non-Reordering, No Early Write Acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement.

ArmV8 added this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

The Device-nGRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

ArmV8 added this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that speculative accesses to Device-GRE memory is forbidden.

The Device-GRE memory type is introduced into the AArch32 translation table formats when the PE is using the Long Descriptor Translation Table format.

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
 - The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.
 - Earlier versions of the Arm architecture defined the following memory types:
 - Strongly-ordered memory. This is the equivalent of the Device-nGnRnE memory type.
 - Device memory. This is the equivalent of the Device-nGnRE memory type.
-

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

An exception to this applies:

- Reads generated by the Advanced SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

Note

- An instruction that generates a sequence of accesses as described in [Atomicity in the Arm architecture](#) might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program defines only a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.
 - Write speculation that is visible to other observers is prohibited for all memory types.
-

- An Explicit Memory Write Effect to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit memory write effect by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Explicit Memory Write Effects to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two Explicit Memory Write Effects of Device memory type to the same location might be merged before they reach the endpoint, unless both Explicit Memory Write Effects have the non-Gathering attribute or there is an ordered-before relationship between the two Explicit Memory Write Effects.
- If a memory location is not capable of supporting unaligned memory accesses, then an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- If a memory location is capable of supporting unaligned memory accesses, and such a memory location is marked as Device, then it is IMPLEMENTATION DEFINED whether an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as execute-never for all Exception levels.

Note

This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as execute-never for all Exception levels is a programming error.

Note

In the PL1&0 translation regime in systems where [HCR.TGE](#) == 1 and [HCR.DC](#) == 0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes the value of [HSR.ISS.\[24\]](#) to be 0.

See also [Memory access restrictions](#).

The memory types for translation table walks cannot be defined as any Device memory type within the TCR. For the EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the [HCR.PTW](#) bit is 1, a stage 2 Permission fault is generated if a first stage translation table walk is made to any Device memory type.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

B2.9.2.1 Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All access occur at their programmed size, except that there is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See [Multi-register loads and stores that access Device memory](#).

Gathering between memory accesses separated by a memory barrier that affects those memory accesses is not permitted.

Gathering between two memory accesses generated by a Load-Acquire/Store-Release is not permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

Note

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
 - The accesses are not separated by a DMB or DSB barrier that affects both of the accesses.
 - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
 - The accesses are not generated by a Store-Release instruction.
 - The Arm architecture defines only programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.
-

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

B2.9.2.2 Reordering

In the Device memory attribute:

R Indicates that the location has the Reordering attribute.

nR Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

Note

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the DMB instruction.
 - The Arm architecture defines only programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.
-

An implementation is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.

An additional relaxation is that an implementation is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

B2.9.2.3 Early Write Acknowledgement

In the Device memory attribute:

E Indicates that the location has the Early Write Acknowledgement attribute.

nE Indicates that the location has the No Early Write Acknowledgement attribute.

If the No Early Write Acknowledgement attribute is assigned for a Device memory location:

- For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgment of a write comes from the endpoint, it is guaranteed that:
 - Only the endpoint of the write access returns a write acknowledgment of the access.
 - No earlier point in the memory system returns a write acknowledgment.
- For memory system endpoints where the system architecture in which the PE is operating does not require that acknowledgment of a write comes from the endpoint, the acknowledgment of a write is not required to come from the endpoint.

Note

A write with the No Early Write Acknowledgement attribute assigned for a Device memory location is not expected to generate an abort in any situation where the equivalent write to the same location without the No Early Write Acknowledgement attribute assigned does not generate an abort.

This means that a `DSB` barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system if that is required by the system architecture.

Peripherals are an example of system endpoints that require that the acknowledgment of a write comes from the endpoint.

Note

- The Early Write Acknowledgement attribute affects only where the endpoint acknowledgment is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.
 - The areas of the physical memory map for which write acknowledgment from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgment from the endpoint.
 - Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgment from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgment from the endpoint.
 - For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a `DSB` instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgment from the endpoint.
-

B2.9.2.4 Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load and store instructions.

For all instructions that load or store one or more registers from the SIMD and floating-point register file there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load and store instructions.

The architecture permits that the non-speculative execution of an instruction that loads or stores more than one general-purpose or SIMD and floating-point register might result in repeated accesses to the same address, even if the resulting accesses are to any type of Device memory.

For an `LDRD`, `STRD`, or `LDM` instruction with a register list that includes the PC, or an `STM` instruction with a register list that includes the PC, the order in which the registers are accessed is not defined by the architecture.

For a load or store of an Advanced SIMD element or structure, the order in which the registers are accessed is not defined by the architecture.

For a `VLDM` and `VSTM` instruction with a register list that does not include the PC, all registers are accessed in ascending address order for accesses to Device memory with the non-Reordering attribute.

For a LDM or STM instruction with a register list that does not include the PC, all registers are accessed in ascending address order for accesses to Device memory with the non-Reordering attribute.

B2.9.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For two explicit memory read effects to any two adjacent bytes in memory, p and $p+1$, generated by the same instruction, and for two explicit memory write effects to any two adjacent bytes in memory, p and $p+1$, that are generated by the same instruction:
 - The bytes p and $p+1$ must have the same memory type and Shareability attributes. otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an LDC, LDM, LDRD STC, STM or STRD instruction, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
 - Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes p and $p+1$.

For the permitted CONSTRAINED UNPREDICTABLE behavior, see [Crossing a page boundary with different memory types or Shareability attributes](#).

- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross a 4KB address boundary then behavior is CONSTRAINED UNPREDICTABLE and [Crossing a 4KB boundary with a Device access](#) describes the permitted behaviors.

Note

- The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.
 - This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses a 4KB address boundary.
 - Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.
-

B2.10 Mismatched memory attributes

Mismatched memory attributes are controlled by privileged software.

Physical memory Locations are accessed with *mismatched attributes* if all accesses to the Location do not use a common definition of all of the following attributes of that Location:

- Memory type: Device-nGnRnE, Device-nGnRE, Device-nGRE, Device-GRE or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

Note

In this document, the terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes, the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
 - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
 - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of speculative read accesses.
- Prohibition on Gathering.
- Prohibition on Reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
 - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
 - Either:
 - All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable or Write-Through.
 - All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.
2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
 - Any Device memory type.
 - Any Normal, Inner Non-cacheable or Write-Through, Outer Non-cacheable or Write-Through memory.

In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:

- Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
- Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting DMB barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory Location all assign the same Shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a Shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
 - Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
 - After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
 - Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
 - Executing a DMB barrier instruction between any accesses to the same cacheable Location that use different attributes.

Note

In AArch32 state, cache maintenance instructions can be accessed only from an Exception level that is higher than EL0, and therefore require a system call. For information on system calls, see [Exception-generating and exception-handling instructions](#). For information about the AArch32 cache maintenance instructions, see [AArch32 cache and branch predictor support](#).

In all cases:

- Location refers to any byte within the current coherency granule.
- A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
- In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see [Ordering of cache and branch predictor maintenance instructions](#).

Note

With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different Shareability attributes, then ordering and coherency are guaranteed only if:
 - Software running on a PE cleans and invalidates a Location from cache before and after each read or write to that Location by that PE.
 - A DMB barrier is placed between any accesses to the same memory Location that use different attributes.

Note

The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

Note

As described in Non-cacheable accesses and instruction caches, a non-cacheable access is permitted to be cached in an instruction cache, despite the fact that a non-cacheable access is not permitted to be cached in a unified cache. Despite this, when cacheable and non-cacheable aliases exist for memory which is executable, these must be treated as mismatched aliases to avoid coherency issues from the data or unified caches that might hold entries that will be brought into the instruction caches.

B2.11 Synchronization and semaphores

The architecture provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal and Device memory.

Note

Use of the synchronization primitives scales for multiprocessing system designs.

Table B2-5 shows the synchronization primitives and the associated CLREX instruction.

Table B2-5 Synchronization primitives and associated instruction, T32 and A32 instruction sets

Transaction size	Additional semantics	Load-Exclusive ^a	Store-Exclusive ^a	Other ^a
Byte	-	LDREXB	STREXB	-
	Load-Acquire/Store-Release	LDAEXB	STLEXB	-
Halfword	-	LDREXH	STREXH	-
	Load-Acquire/Store-Release	LDAEXH	STLEXH	-
Word	-	LDREX	STREX	-
	Load-Acquire/Store-Release	LDAEX	STLEX	-
Doubleword	-	LDREXD	STREXD	-
	Load-Acquire/Store-Release	LDAEXD	STLEXD	-
None	Clear-Exclusive	-	-	CLREX

a. Instruction in the T32 and A32 instruction sets.

Except for the row showing the CLREX instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address x is:

- The Load-Exclusive instruction reads a value from memory address x .
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address x only if no other observer, process, or thread has performed a more recent store to address x . The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see [Marking and the size of the marked memory block](#). A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- [Exclusive access instructions and Non-shareable memory locations](#).
- [Exclusive access instructions and shareable memory locations](#).
- [Marking and the size of the marked memory block](#).
- [Context switch support](#).
- [Load-Exclusive and Store-Exclusive instruction usage restrictions](#).
- [Use of WFE and SEV instructions by spin-locks](#).

B2.11.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the Shareability attribute is Non-shareable, the exclusive access instructions rely on a *local Exclusives monitor*, or *local monitor*, that marks any address from which the PE executes a Load-Exclusive instruction. Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
 - If the store took place the status value is 0.
 - Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state the monitor is *set*.

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Exclusive Access state the monitor is *clear*:

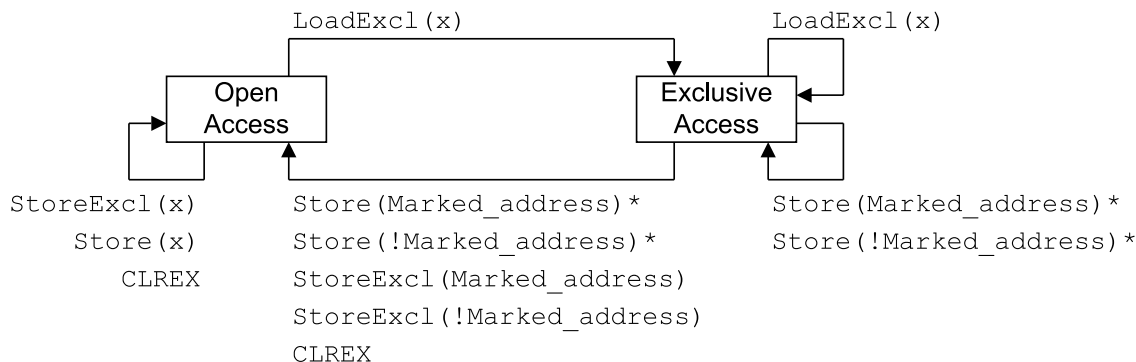
The Store-Exclusive instruction defines the register to which the status value is returned.

When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

[Figure B2-4](#) shows the state machine for the local monitor and the effect of each of the operations shown in the figure.



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: `LoadExcl` represents any Load-Exclusive instruction

`StoreExcl` represents any Store-Exclusive instruction

`Store` represents any other store instruction.

Any `LoadExcl` operation updates the marked address to the most significant bits of the address `x` used for the operation.

Figure B2-4 Local monitor state machine diagram

For more information about marking, see [Marking and the size of the marked memory block](#).

Note

For the local monitor state machine, as shown in [Figure B2-4](#):

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction, by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

B2.11.1.1 Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in [Figure B2-4](#).

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in [Figure B2-4](#).
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in [Figure B2-4](#) must not indefinitely delay forward progress of execution.

B2.11.2 Exclusive access instructions and shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, which marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in [Exclusive access instructions and Non-shareable](#)

[memory locations](#), except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:

- Updating memory.
- Returning a status value of 0.

The local monitor can ignore accesses from other PEs in the system.

- A *global monitor* that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the Shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:
 - Can hold at least one marked block.
 - Maintains a state machine for each marked block it can hold.

Note

For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is CONSTRAINED UNPREDICTABLE, see [Load-Exclusive and Store-Exclusive instruction usage restrictions](#).

Note

The global monitor can either reside in a block that is part of the hardware on which the PE executes or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

Note

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive/Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.
- The instruction generates an IMPLEMENTATION DEFINED [MPU fault](#). This is reported using the Fault status code of:

- **DFSR.STATUS** = 0b110101. The fault can also be reported in the **HSR.ISS**[5:0] field for exceptions to Hyp mode.

If the IMPLEMENTATION DEFINED **MPU fault** is generated for the PL1&0 translation regime then:

- If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.
- Otherwise, the fault is a second stage fault and the exception is taken to EL2.

The priority of this fault is IMPLEMENTATION DEFINED.

- The instruction is treated as a **NOP**.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes **UNKNOWN**.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes **UNKNOWN**.
- The value held in the result register of the Store-Exclusive instruction becomes **UNKNOWN**.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by an Arm PE is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

B2.11.2.1 Operation of the global Exclusives monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

Note

The global monitor only supports a single outstanding exclusive access to shareable memory for each PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:
 - A status value of 0 is returned to a register to acknowledge the successful store.
 - The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.
 - If the address accessed is marked for exclusive access in the global monitor state machine for any other PE then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
 - A status value of 1 is returned to a register to indicate that the store failed.
 - The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
 - If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
 - If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means it responds to:

- Accesses generated by PE(n).

- Accesses generated by the other observers in the Shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

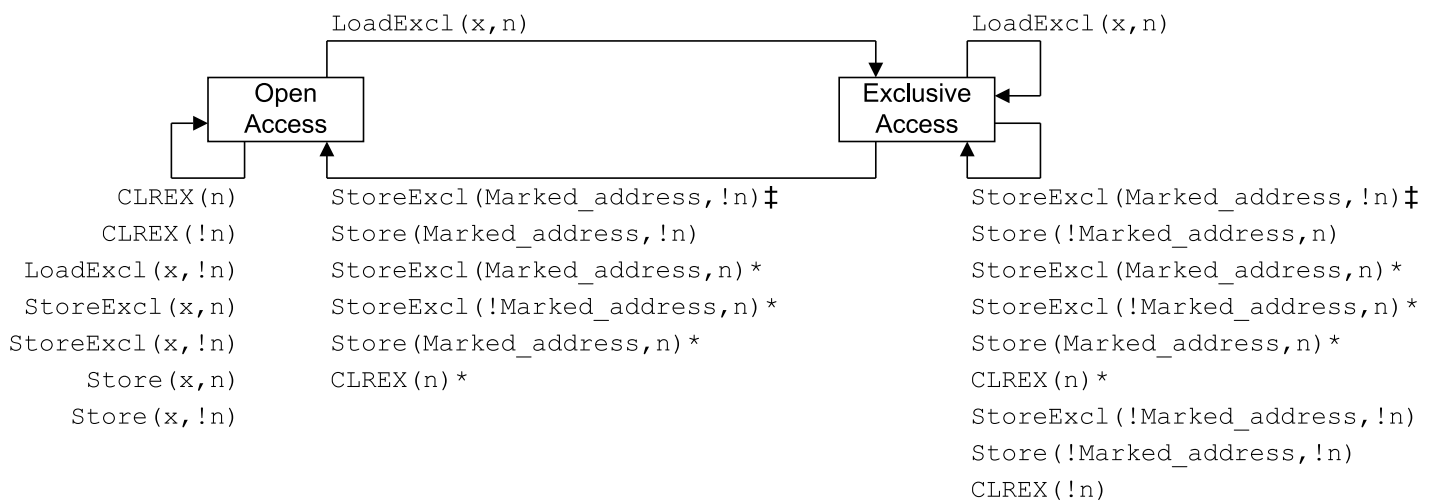
A global monitor:

- In the Exclusive Access state is *set*.
- In the Open Access state is *clear*.

B2.11.2.1.1 Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see [Wait For Event and Send Event](#).

Figure B2-5 shows the state machine for PE(n) in a global monitor.



‡StoreExcl(Marked_address, !n) clears the monitor only if the StoreExcl updates memory

Operations marked * are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: LoadExcl represents any Load-Exclusive instruction

StoreExcl represents any Store-Exclusive instruction

Store represents any other store instruction.

Any LoadExcl operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B2-5 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking, see [Marking and the size of the marked memory block](#).

Note

For the global monitor state machine, as shown in [Figure B2-5](#):

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
 - Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, [Figure B2-5](#) shows only how the operations by (!n) cause state transitions of the state machine for PE(n).
 - A Load-Exclusive instruction can update only the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
 - When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a CLREX instruction causes the global monitor to transition from Exclusive Access to Open Access state.
 - It is IMPLEMENTATION DEFINED:
 - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
 - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.
-

B2.11.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size 2^a bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4 - 512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where a is 4.
- 512 words in an implementation where a is 11.

For example, in an implementation where a is 4, a successful LDREXB of address $0 \times 341B4$ defines a marked block using bits[47:4] of the address. This means that the four words of memory from $0 \times 341B0$ to $0 \times 341BF$ are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see [CTR](#). Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

B2.11.4 Context switch support

An exception return clears the local monitor. As a result, performing a CLREX instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

B2.11.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a LDREX/STREX pair or a LDREXB/STREXB pair, executed as part of a loop that contains only a single Load-Exclusive and

Store-Exclusive pair. To support different implementations of these functions, software must follow the notes and restrictions given in this subsection.

The following notes describe use of a Load-Exclusive/ Store-Exclusive instruction pair, `LoadExcl/StoreExcl`, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table B2-5. In this context, a `LoadExcl/StoreExcl` pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target VA of a `StoreExcl` is different from the VA of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
 - The `StoreExcl` either passes or fails, the status value returned by the `StoreExcl` is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

———— **Note** ————

This means the `StoreExcl` might pass for some instances of a `LoadExcl/StoreExcl` pair with mismatched addresses, and fail for other instances of a `LoadExcl/StoreExcl` pair with mismatched addresses.

- The data at the address accessed by the `LoadExcl`, and at the address accessed by the `StoreExcl`, is UNKNOWN.

This means software can rely on a `LoadExcl/StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` are executed with the same VA.

- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a `StoreExcl` instruction is the same as the transaction size of the preceding `LoadExcl` instruction executed in that thread. If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
 - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

———— **Note** ————

This means the `StoreExcl` might pass for some instances of a `LoadExcl/StoreExcl` pair with mismatched transaction sizes, and fail for other instances of a `LoadExcl/StoreExcl` pair with mismatched transaction sizes.

- The block of data of the size of the larger of the transaction sizes used by the `LoadExcl/StoreExcl` pair at the address accessed by the `LoadExcl/StoreExcl` pair, is UNKNOWN.

This means software can rely on a `LoadExcl/StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` have the same transaction size.

- `LoadExcl/StoreExcl` loops are guaranteed to make forward progress only if, for any `LoadExcl/StoreExcl` loop within a single thread of execution, the software meets all of the following conditions.
- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.
- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.
- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.
- For the memory location accessed by a `LoadExcl/StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:
 - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

———— **Note** ————

This means the `StoreExcl` might pass for some instances of a `LoadExcl/StoreExcl` pair with changed memory attributes, and fail for other instances of a `LoadExcl/StoreExcl` pair with changed memory attributes.

- The data at the address accessed by the `StoreExcl` is UNKNOWN.

Note

Another bullet point in this list covers the case where the memory attributes of a `LoadExcl/StoreExcl` pair differ as a result of using different VAs with different attributes that point to the same PA.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same Shareability domain as the PE executing the cache maintenance instruction, as determined by the Shareability domain of the address being maintained.
- If the mapping of the VA to PA is changed between the `LoadExcl` instruction and the `StoreExcl` instruction, and the change is performed using a break-before-make sequence as described in [Using break-before-make when updating translation table entries](#), if the `StoreExcl` is performed after another write to the same PA as the `StoreExcl`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `StoreExcl` will not pass its monitor check.

Note

Arm expects that, in many implementations, either:

- The TLB invalidation will clear either the local or global monitor.
 - The PA will be checked between the `LoadExcl` and `StoreExcl`.
-

- The Exclusive Access state for an address accessed by a PE can be lost as a result of a `PLDW` instruction to the same PA executed by another PE. This means that a very high rate of repeated `PLDW` accesses to a memory location might impede the forward progress of another PE.

Note

In the event of repeatedly-contending `LoadExcl/StoreExcl` instruction sequences from multiple PEs, an implementation must ensure that forward progress is made by at least one PE.

B2.11.6 Use of WFE and SEV instructions by spin-locks

The architecture provides Wait For Event, Send Event, and Send Event Local instructions, `WFE`, `SEV`, `SEVL`, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in [Wait For Event and Send Event](#). However, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note

This is equivalent to issuing an `SEVL` instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an `SEV` instruction after clearing a spinlock.

Part C The AArch32 Instruction Sets

Chapter C1

About the T32 and A32 Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- [Format of instruction descriptions](#)
- [Standard assembler syntax fields](#)
- [Conditional execution](#)
- [Shifts applied to a register](#)
- [Memory accesses](#)
- [Encoding of lists of general-purpose registers and the PC](#)
- [General information about the T32 and A32 instruction descriptions](#)
- [Additional pseudocode support for instruction descriptions](#)
- [Additional information about Advanced SIMD and floating-point instructions](#)

C1.1 Format of instruction descriptions

The instruction descriptions in [T32 and A32 Base Instruction Set Instruction Descriptions](#) and [T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions](#) normally use the following format:

- Instruction section title.
- Introduction to the instruction.
- A description of each encoding of the instruction.
- Assembler syntax.
- Pseudocode describing how the instruction operates.
- Notes, if applicable.

Each of these items is described in more detail in the following subsections.

C1.1.1 Instruction section title

The instruction section title gives the base mnemonic for the instruction or instructions described in the section. When one mnemonic has multiple forms described in separate instruction sections, this is followed by a short description of the form in parentheses. The most common use of this is to distinguish between forms of an instruction in which one of the operands is an immediate value and forms in which it is a register.

C1.1.2 Introduction to the instruction

The introduction to the instruction briefly describes the main features of the instruction. This description is not necessarily complete and is not definitive. If there is any conflict between it and the more detailed information that follows, the latter takes priority.

C1.1.3 Instruction encodings

This is a list of one or more instruction encodings. Each instruction encoding is labeled as:

- A1, A2, A3 . . . for the first, second, third, and any additional A32 encodings.
- T1, T2, T3 . . . for the first, second, third, and any additional T32 encodings.

Each instruction encoding description consists of:

- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. Sometimes, multiple syntax variants are given. These are written in a `typewriter` font using the conventions described in [Assembler syntax prototype line conventions](#). The correct one to use can be indicated by:
 - A subheading that identifies the encodings that correspond to the syntax. See, for example, the subheading *Flag setting, rotate right with extend variant* in the description of the A1 encoding of the ADC, ADCS (register) instructions in [A1](#).
 - An annotation to the syntax, such as *Inside IT block* or *Outside IT block*. See, for example, the syntax descriptions of the T1 encoding of the ADC, ADCS (register) instructions in [T1](#).

In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

There is usually more than one syntax variant that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants, and other operands to the instruction. For example, when assembling to the T32 instruction set, the syntax `AND R0, R0, R8` ensures selection of a 32-bit encoding but `AND R0, R0, R1` selects a 16-bit encoding.

For each instruction encoding belonging to a target instruction set, an assembler can use this information to determine whether it can use that encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction, then:

- If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecture *prefers* the 16-bit encoding. This means the assembler must use the 16-bit encoding rather than the 32-bit encoding. Software can use the `.W` and `.N` qualifiers to specify the required encoding width, see [Standard assembler syntax fields](#).

- If multiple encodings of the same length can encode the instruction, the *Assembler syntax* subsection says which encoding is preferred, and how software can, instead, select the other encodings. Each encoding also documents UAL syntax that selects it in preference to any other encoding.

If no encodings of the target instruction set can encode the instruction requested by the UAL source, normally the assembler generates an error saying that the instruction is not available in that instruction set.

Note

In some cases, an instruction is available in one instruction set but not in another. The *Assembler syntax* subsection identifies many of these cases. For example, the A32 instructions with bits<31:28> == 0b1111 described in [Branch, branch with link, and block data transfer](#), [System register access](#), [Advanced SIMD, floating-point, and Supervisor call](#), and [Unconditional instructions](#) cannot have a Condition code, but the equivalent T32 instructions often can, and this usually appears in the *Assembler syntax* subsection as a statement that the A32 instruction cannot be conditional.

However, some such cases are too complex to describe in the available space, so the definitive test of whether an instruction is available in a given instruction set is whether there is an available encoding for it in that instruction set.

The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers might use simpler syntaxes when they are suitable for the operand combination, to produce more readable disassembled code.

- An encoding diagram, where:
 - For a 32-bit A32 encoding diagram, the bits are numbered from 31-0.
 - For a 16-bit T32 encoding diagram, the bits are numbered from 15-0. This halfword can be described as *hw1* of the instruction.
 - For a 32-bit T32 encoding diagram, the bits are numbered from 15-0 for each halfword, as a reminder that a 32-bit T32 instruction consists of two consecutive halfwords rather than a word. In this case, the left-hand halfword in the diagram is identified as *hw1*, and the right-hand halfword is identified as *hw2*.

Where instructions are stored using the standard little-endian instruction endianness:

- The encoding diagram for an A32 instruction at address *A* shows, from left to right, the bytes at addresses *A+3*, *A+2*, *A+1*, *A*.
- The encoding diagram for a 32-bit T32 instruction shows bytes in the order *A+1*, *A* for *hw1*, followed by bytes *A+3*, *A+2* for *hw2*.
- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see [Arm Pseudocode Definition](#).

C1.1.4 Assembler symbols

The *Assembly symbols* describe the standard UAL syntax for the instruction.

Each syntax description consists of the following elements:

- Descriptions of all variable or optional fields of the syntax.

Some syntax fields are standardized across all or most instructions. [Standard assembler syntax fields](#) describes these fields.

By default, syntax fields that specify registers, such as <*Rd*>, <*Rn*>, or <*Rt*>, can be any of R0-R12 or LR in T32 instructions, and any of R0-R12, SP, or LR in A32 instructions. These require that the encoding-specific pseudocode set the corresponding integer variable (such as *d*, *n*, or *t*) to the corresponding register number, using 0-12 for R0-R12, 13 for SP, or 14 for LR:

 - Normally, software can do this by setting the corresponding field in the instruction, typically named *Rd*, *Rn*, *Rt*, to the binary encoding of that number.
 - In the case of 16-bit T32 encodings, the field is normally of length 3, and so the encoding is available only when the assembler syntax specifies one of R0-R7. Such encodings often use a register field name like *Rdn*. This indicates that the encoding is available only if <*Rd*> and <*Rn*> specify the same register, and that the register number of that register is encoded in the field if they do.

The description of a syntax field that specifies a register sometimes extends or restricts the permitted range of registers or documents other differences from the default rules for such fields. Examples of extensions are permitting the use of the SP in a T32 instruction, or permitting the use of the PC, identified using register number 15.

- Where appropriate, text that briefly describes changes from the pre-UAL assembler syntax. Where present, this usually consists of an alternative pre-UAL form of the assembler mnemonic. The pre-UAL assembler syntax does not conflict with UAL. Arm recommends that it is supported, as an optional extension to UAL, so that pre-UAL assembler source files can be assembled.

C1.1.4.1 Assembler syntax prototype line conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

< > Any item bracketed by < and > is a short description of a type of value to be supplied by the user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence requires only the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for an instruction contains an item <Rn> and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded. This is often done by specifying a required output from the encoding-specific pseudocode, such as `add = TRUE`. The assembler must use only encodings that produce that output.

{ } Any item bracketed by { and } is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text.

Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.

In the assembler syntax, numeric constants are normally preceded by a #. Some UAL instruction syntax descriptions explicitly show this # as optional. Any UAL assembler:

- Must treat the # as optional where an instruction syntax description shows it as optional.
- Can treat the # either as mandatory or as optional where an instruction syntax description does not show it as optional.

Note

Arm recommends that UAL assemblers treat all uses of # shown in this manual as optional.

spaces Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

+/- This indicates an optional + or - sign. If neither is coded, + is assumed.

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. In a few places, the { and } characters must be encoded as part of a variable item. When this happens, the long description of the variable item indicates how they must be used.

C1.1.5 Pseudocode describing how the instruction operates

The *Operation for all classes* subsection contains encoding-independent pseudocode that describes the main operation of the instruction.

C1.1.5.1 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses, see [External visibility requirement](#).
- Pseudocode does not describe the exact rules when an instruction that generates any of the following fails its condition code check:
 - UNDEFINED instruction.
 - Hyp trap.

In such cases, the `Undefined()` pseudocode statement or call to the applicable trap function lies inside the `if ConditionPassed() then ...` structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. For the exact rules, see:

- [Conditional execution of undefined instructions](#).
- [EL2 configurable controls](#).
- Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped. [Combinations of floating-point exceptions](#) describes the exact rules.

Note

There is no limitation in the case where all the floating-point exceptions are untrapped, because the pseudocode specifies the same behavior as the cross-referenced section.

- An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `Abort()`, or implicitly, for example if an interrupt is taken during execution of an LDM instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs.

C1.1.5.2 Instruction encoding diagrams and instruction pseudocode

This is a list of one or more instruction encodings. Each instruction encoding is labeled as:

- A1, A2, A3 ... for the first, second, third, and any additional A32 encodings.
- T1, T2, T3 ... for the first, second, third, and any additional T32 encodings.

Each instruction encoding description consists of:

- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. Sometimes, multiple syntax variants are given. These are written in a *typewriter* font using the conventions described in [Assembler syntax prototype line conventions](#). The correct one to use can be indicated by:
 - A subheading that identifies the encodings that correspond to the syntax. See, for example, the subheading *Flag setting, rotate right with extend variant* in the description of the A1 encoding of the ADC, ADCS (register) instructions in [A1](#).
 - An annotation to the syntax, such as *Inside IT block* or *Outside IT block*. See, for example, the syntax descriptions of the T1 encoding of the ADC, ADCS (register) instructions in [T1](#).

In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

There is usually more than one syntax variant that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants, and other operands to the instruction. For example, when assembling to the T32 instruction set, the syntax `AND R0, R0, R8` ensures selection of a 32-bit encoding but `AND R0, R0, R1` selects a 16-bit encoding.

For each instruction encoding belonging to a target instruction set, an assembler can use this information to determine whether it can use that encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction, then:

- If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecture *prefers* the 16-bit encoding. This means the assembler must use the 16-bit encoding rather than the 32-bit encoding. Software can use the `.W` and `.N` qualifiers to specify the required encoding width, see [Standard assembler syntax fields](#).
- If multiple encodings of the same length can encode the instruction, the *Assembler syntax* subsection says which encoding is preferred, and how software can, instead, select the other encodings. Each encoding also documents UAL syntax that selects it in preference to any other encoding.

If no encodings of the target instruction set can encode the instruction requested by the UAL source, normally the assembler generates an error saying that the instruction is not available in that instruction set.

Note

In some cases, an instruction is available in one instruction set but not in another. The *Assembler syntax* subsection identifies many of these cases. For example, the A32 instructions with bits<31:28> == 0b1111 described in [Branch, branch with link, and block data transfer](#), [System register access](#), [Advanced SIMD, floating-point, and Supervisor call](#), and [Unconditional instructions](#) cannot have a Condition code, but the equivalent T32 instructions often can, and this usually appears in the *Assembler syntax* subsection as a statement that the A32 instruction cannot be conditional.

However, some such cases are too complex to describe in the available space, so the definitive test of whether an instruction is available in a given instruction set is whether there is an available encoding for it in that instruction set.

The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers might use simpler syntaxes when they are suitable for the operand combination, to produce more readable disassembled code.

- An encoding diagram, where:
 - For a 32-bit A32 encoding diagram, the bits are numbered from 31-0.
 - For a 16-bit T32 encoding diagram, the bits are numbered from 15-0.
This halfword can be described as `hw1` of the instruction.
 - For a 32-bit T32 encoding diagram, the bits are numbered from 15-0 for each halfword, as a reminder that a 32-bit T32 instruction consists of two consecutive halfwords rather than a word.
In this case, the left-hand halfword in the diagram is identified as `hw1`, and the right-hand halfword is identified as `hw2`.

Where instructions are stored using the standard little-endian instruction endianness:

- The encoding diagram for an A32 instruction at address `A` shows, from left to right, the bytes at addresses `A+3`, `A+2`, `A+1`, `A`.
- The encoding diagram for a 32-bit T32 instruction shows bytes in the order `A+1`, `A` for `hw1`, followed by bytes `A+3`, `A+2` for `hw2`.
- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see [Instruction encoding diagrams and instruction pseudocode](#).

C1.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

<c> Is an optional field. It specifies the condition under which the instruction is executed. See [Conditional execution](#) for the range of available conditions and their encoding. If <c> is omitted, it defaults to *always* (AL).

<q> Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

.N Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

.W Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither .W nor .N is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description. The assembler syntax includes a mandatory .W qualifier, along with a note describing the cases in which it applies, where this qualifier is required to select a particular encoding for an instruction. Additional assembler syntax will describe the syntax when the conditions are not met.

Note

When assembling to the A32 instruction set, the .N qualifier produces an assembler error and the .W qualifier has no effect.

C1.3 Conditional execution

Most T32 and A32 instructions can be executed conditionally, based on the values of the APSR Condition flags. [Table C1-1](#) lists the available conditions.

Table C1-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) ^a	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS ^b	Carry set	Greater than, equal, or unordered	C == 1
0011	CC ^c	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) ^d	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.

b. HS (unsigned higher or same) is a synonym for CS.

c. LO (unsigned lower) is a synonym for CC.

d. AL is an optional mnemonic extension for always, except in IT instructions. For details, see IT.

In T32 instructions, the condition, if it is not AL, is normally encoded in a preceding IT instruction. For more information, see [Conditional instructions](#) and [IT](#). Some conditional branch instructions do not require a preceding IT instruction, because they include a Condition code in their encoding.

Implementations can provide a set of ITD control fields, [SCTLR.ITD](#) and [HSCTLR.ITD](#), to disable use of IT for some instructions, making them UNDEFINED.

In A32 instructions, bits[31:28] of the instruction contain either:

- The Condition code, see [The Condition code field in A32 instruction encodings](#).
- 0b1111 for some A32 instructions that can only be executed unconditionally.

C1.3.1 The Condition code field in A32 instruction encodings

Every conditional A32 instruction contains a 4-bit Condition code field, the cond field, in bits 31-28:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond																															

This field contains one of the values 0b0000-0b1110, as shown in [Table C1-1](#). Most instruction mnemonics can be extended with the letters defined in the [Mnemonic extension](#) column of that table.

If the *always* (AL) condition is specified, the instruction is executed irrespective of the value of the Condition flags. The absence of a Condition code on an instruction mnemonic implies the AL Condition code.

C1.3.2 Pseudocode description of conditional execution

The `CurrentCond()` function returns a 4-bit condition specifier as follows:

- For A32 instructions, it returns bits[31:28] of the instruction.
- For the T1 and T3 encodings of the Branch instruction (see [B](#)), it returns the 4-bit `cond` field of the encoding.
- For all other T32 instructions:
 - If `PSTATE.IT<3:0> != '0000'` it returns `PSTATE.IT<7:4>`.
 - If `PSTATE.IT<7:0> == '00000000'` it returns `'1110'`.
 - Otherwise, execution of the instruction is `CONSTRAINED UNPREDICTABLE`.

For more information, see [Process state, PSTATE](#).

The `ConditionPassed()` function uses this condition specifier and the Condition flags to determine whether the instruction must be executed, by calling the `ConditionHolds()` function.

[A-profile Architecture Pseudocode](#) includes the definitions of these functions.

[Undefined Instruction exception](#) describes the handling of conditional instructions that are `UNDEFINED`, `UNPREDICTABLE`, or `CONSTRAINED UNPREDICTABLE`. The pseudocode in the manual, as a sequential description of the instructions, has limitations in this respect. For more information, see [Limitations of the instruction pseudocode](#).

C1.4 Shifts applied to a register

A32 register offset load/store word and unsigned byte instructions can apply a wide range of different constant shifts to the offset register. Both T32 and A32 data-processing instructions can apply the same range of different constant shifts to the second operand register. For details, see [Constant shifts](#).

A32 data-processing instructions can apply a register-controlled shift to the second operand register.

C1.4.1 Constant shifts

These are the same in T32 and A32 instructions, except that the input bits come from different positions.

`<shift>` is an optional shift to be applied to `<Rm>`. It can be any one of:

(omitted)	No shift.
LSL #<n>	Logical shift left <n> bits. $1 \leq \text{<n>} \leq 31$.
LSR #<n>	Logical shift right <n> bits. $1 \leq \text{<n>} \leq 32$.
ASR #<n>	Arithmetic shift right <n> bits. $1 \leq \text{<n>} \leq 32$.
ROR #<n>	Rotate right <n> bits. $1 \leq \text{<n>} \leq 31$.
RRX	Rotate right one bit, with extend. Bit[0] is written to <code>shifter_carry_out</code> , bits[31:1] are shifted right one bit, and the Carry flag is shifted into bit[31].

Note

Assemblers can permit the use of some or all of ASR #0, LSL #0, LSR #0, and ROR #0 to specify that no shift is to be performed. This is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must omit the shift specifier when the instruction specifies no shift.

Similarly, assemblers can permit the use of #0 in the immediate forms of ASR, LSL, LSR, and ROR instructions to specify that no shift is to be performed, that is, that a MOV (register) instruction is wanted. Again, this is not standard UAL, and the encoding selected for T32 instructions might vary between UAL assemblers if it is used. To ensure disassembled code assembles to the original instructions, disassemblers must use the MOV (register) syntax when the instruction specifies no shift.

C1.4.1.1 Encoding

The assembler encodes `<shift>` into two type bits and five immediate bits, as follows:

(omitted)	type = 0b00, immediate = 0.
LSL #<n>	type = 0b00, immediate = <n>.
LSR #<n>	type = 0b01. If <n> < 32, immediate = <n>. If <n> == 32, immediate = 0.
ASR #<n>	type = 0b10. If <n> < 32, immediate = <n>. If <n> == 32, immediate = 0.
ROR #<n>	type = 0b11, immediate = <n>.
RRX	type = 0b11, immediate = 0.

C1.4.2 Register controlled shifts

These are available only in A32 instructions.

<type> is the type of shift to apply to the value read from <Rm>. It must be one of:

ASR	Arithmetic shift right, encoded as type = 0b10.
LSL	Logical shift left, encoded as type = 0b00.
LSR	Logical shift right, encoded as type = 0b01.
ROR	Rotate right, encoded as type = 0b11.

The bottom byte of <Rs> contains the shift amount.

C1.4.3 Pseudocode description of instruction-specified shifts and rotates

The pseudocode enumeration `SRType` { } defines the shift types. Shift and rotate instruction decode is described by the pseudocode function:

- `DecodeImmShift` () for a constant shift.
- `DecodeRegShift` () for a register controlled shift.

Shift and rotate operations are made by the pseudocode function `Shift` () .

C1.5 Memory accesses

Commonly, the following addressing modes are permitted for memory access instructions:

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The value of the base register is unchanged.

The assembly language syntax for this mode is:

[<Rn>, <offset>]

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

[<Rn>, <offset>]!

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register.

The assembly language syntax for this mode is:

[<Rn>], <offset>

In each case, <Rn> is the base register. <offset> can be:

- An immediate constant, such as <imm8> or <imm12>.
- An index register, <Rm>.
- A shifted index register, such as <Rm>, LSL #<shift>.

For information about unaligned access, endianness, and exclusive access, see:

- [Alignment support](#).
- [Endian support](#).
- [Synchronization and semaphores](#).

C1.6 Encoding of lists of general-purpose registers and the PC

A number of instructions operate on lists of general-purpose registers. For some load instructions, the list of registers to be loaded can include the PC. For these instructions, the assembler syntax includes a `<registers>` field, which provides a list of the registers to be operated on, with list entries separated by commas.

The registers list is encoded in the instruction encoding. Most often, this is done using an 8-bit, 13-bit, or 16-bit `register_list` field. This section gives more information about these and other possible register list encodings.

In a `register_list` field, each bit corresponds to a single register, and if the `<registers>` field of the assembler instruction includes `Rt` then `register_list<t>` is set to 1, otherwise it is set to 0.

The full rules for the encoding of lists of general-purpose registers, and possibly the PC, are:

- Except for the cases listed here, 16-bit T32 encodings use an 8-bit register list, and can access only registers R0-R7.
The exceptions to this rule are:
 - The T1 encoding of `POP` uses an 8-bit register list, and an additional bit, `P`, that corresponds to the PC. This means it can access any of R0-R7 and the PC.
 - The T1 encoding of `PUSH` uses an 8-bit register list, and an additional bit, `M`, that corresponds to the LR. This means it can access any of R0-R7 and the LR.
- 32-bit T32 encodings of load operations use a 13-bit register list, and two additional bits, `M`, corresponding to the LR, and `P`, corresponding to the PC. This means these instructions can access any of R0-R12 and the LR and PC.
- 32-bit T32 encodings of store operations use a 13-bit register list, and one additional bit, `M`, corresponding to the LR. This means these instructions can access any of R0-R12 and the LR.
- Except for the case listed here, A32 encodings use a 16-bit register list. This means these instructions can access any of R0-R12 and the SP, LR, and PC.
The exception to this rule is:
 - The System instructions `LDM` (exception return) and `LDM` (User registers) use a 15-bit register list. This means these instructions can access any of R0-R12 and the SP and LR.
- The T3 and A2 encodings of `POP`, and the T3 and A2 encodings of `PUSH`, access a single register from the set of registers {R0-R12, LR, PC} and encode the register number in the `Rt` field.

Note

`POP` is a load operation, and `PUSH` is a store operation.

In every case, the encoding-specific pseudocode converts the register list into a 32-bit variable, `registers`, with a bit corresponding to each of the registers R0-R12, SP, LR, and PC.

Note

Some Advanced SIMD and floating-point instructions operate on lists of SIMD and floating-point registers. The assembler syntax of these instructions includes a `<list>` field that specifies the registers to be operated on, and the description of the instruction in [Alphabetical list of T32 and A32 base instruction set instructions](#) defines the use and encoding of this field.

C1.7 General information about the T32 and A32 instruction descriptions

[T32 Instruction Set Encoding](#) describes the T32 instruction encodings, and [A32 Instruction Set Encoding](#) describes the A32 instruction encodings. The following subsections give more information about the descriptions of these instructions and their encodings:

- [Execution of instructions in debug state.](#)
- [Fixed values in AArch32 instruction and System register descriptions.](#)
- [UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space.](#)
- [T32 and A32 Advanced SIMD and floating-point instruction encodings.](#)
- [The PC and the use of 0b1111 as a register specifier in T32 and A32 instructions.](#)
- [The SP and the use of 0b1101 as a register specifier in T32 and A32 instructions.](#)
- [Modified immediate constants in T32 and A32 instructions.](#)

C1.7.1 Execution of instructions in debug state

In general, except for the instructions described in [Debug state](#), the T32 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see [Executing instructions in Debug state](#).

-
- Note**
- A32 instructions cannot be executed in Debug state.
 - For many T32 instructions, execution is unchanged in Debug state. [Executing instructions in Debug state](#) identifies these instructions.
-

C1.7.2 Fixed values in AArch32 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The [Glossary](#) gives full descriptions of these terms, and each entry in this section includes a link to the corresponding [Glossary](#) entry.

-
- Note**
- In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the [Glossary](#).
-

The following terms are used to describe bits or fields with fixed values:

RAZ Read-As-Zero. See [Read-As-Zero \(RAZ\)](#).

In diagrams, a RAZ bit can be shown as 0.

(0), RES0 Reserved, [Should-Be-Zero \(SBZ\)](#) or RES0.

In instruction encoding diagrams, and sometimes in other descriptions, (0) indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either (0) (0) (0) or as (000).

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. For more information, see the [Glossary](#) definition of RES0.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,

The (0) and RES0 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The [Glossary](#) definitions cover these cases.

RAO Read-As-One. See [Read-As-One \(RAO\)](#).

In diagrams, a RAO bit can be shown as 1.

(1), RES1 Reserved, [Should-Be-One \(SBO\)](#) or [RES1](#).

In instruction encoding diagrams, and sometimes in other descriptions, (1) indicates an SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either (1) (1) (1) or as (111).

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. For more information, see the [Glossary](#) definition of [RES1](#).

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES1 fields.

The (1) and RES1 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The [Glossary](#) definitions cover these cases.

Note

In register diagrams, (0) is a synonym for RES0, and (1) is a synonym for RES1, where RES0 and RES1 are defined in the [Glossary](#). However, when used in an instruction encoding diagram, (0) and (1) have the narrower definition that behavior is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE if either:

- A bit marked as (0) has the value 1.
 - A bit marked as (1) has the value 0.
-

C1.7.3 UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE. From the introduction of Armv8-R, the architecturally UNPREDICTABLE behavior in AArch32 state is greatly reduced. Most cases that earlier versions of the architecture describe as UNPREDICTABLE become either:
 - CONSTRAINED UNPREDICTABLE, meaning the architecture defines a limited range of permitted behaviors.
 - Fully predictable.
 For more information, see [Architectural Constraints on UNPREDICTABLE Behaviors](#).
- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in [T32 Instruction Set Encoding](#) or [A32 Instruction Set Encoding](#).

An instruction is UNPREDICTABLE only if:

- It is declared as UNPREDICTABLE in an instruction description or in [T32 Instruction Set Encoding](#) or [A32 Instruction Set Encoding](#), and [Architectural Constraints on UNPREDICTABLE Behaviors](#) does not redefine the behavior as CONSTRAINED UNPREDICTABLE.
- The pseudocode for that encoding does not indicate that a different special case applies, and a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively. In most cases, Armv8 makes these cases CONSTRAINED UNPREDICTABLE, as described in [SBZ or SBO fields T32 and A32 in instructions](#).

Unless otherwise specified, T32 and A32 instructions provided as part of an architectural extension, or by an optional feature of the architecture, are UNDEFINED in an implementation that does not include that extension or feature.

Note

Examples of where this rule applies are:

- The instructions provided by the Cryptographic Extension.
 - The System instructions that provide access to the System registers of the OPTIONAL Performance Monitors Extension.
 - The Advanced SIMD and floating-point instructions.
-

For more information about UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction behavior, see [Undefined Instruction exception](#).

C1.7.4 T32 and A32 Advanced SIMD and floating-point instruction encodings

The T32 and A32 encodings of Advanced SIMD and floating-point instructions that are described in [T32 Instruction Set Encoding](#) and in [A32 Instruction Set Encoding](#) are common to the T32 and A32 instruction sets. This means:

- The instruction groups, and the set of instructions in each group, are identical for T32 and A32.
- For each instruction:
 - Each T32 encoding is exactly equivalent to an A32 encoding.
 - There is no T32 encoding without an equivalent A32 encoding, and no A32 encoding without an equivalent T32 encoding.

Note

- In the T32 instruction sets, the Advanced SIMD and floating-point instructions have 32-bit encodings.
 - In the base instruction sets, some instructions are common to the T32 and A32 instruction sets, whereas other instructions have equivalent but not identical functionality in the two instruction sets.
-

32-bit T32 encodings are described as two contiguous halfwords, {hw1:hw2}, as described in [Instruction encodings](#). In general:

- hw1 of a T32 encoding maps onto bits[31:16] of an equivalent A32 encoding.
- hw2 of a T32 encoding maps onto bits[15:0] of an equivalent A32 encoding.

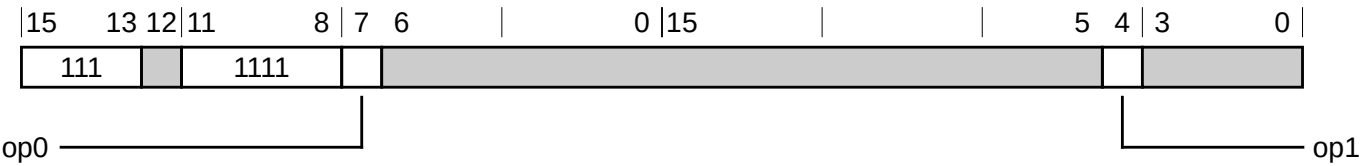
However, the different structures of the T32 instruction encoding space and the A32 instruction encoding space mean that:

- For a given Advanced SIMD and floating-point instruction group:
 - The positions of the fields that identify the instruction, or instruction encoding, within the instruction group might differ between the T32 encodings and the A32 encodings.
 - However, the field values that identify the instruction of instruction encoding are identical for the T32 encoding and the A32 encoding.

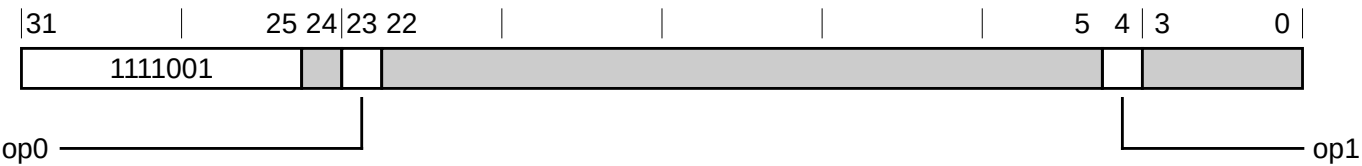
The remainder of this section describes the equivalence of the T32 and A32 encodings for each of the Advanced SIMD and floating-point instruction groups.

C1.7.4.1 Advanced SIMD data-processing

The T32 encoding of the Advanced SIMD data-processing group is:



The A32 encoding of the Advanced SIMD data-processing group is:



The encodings in this group are identified by:

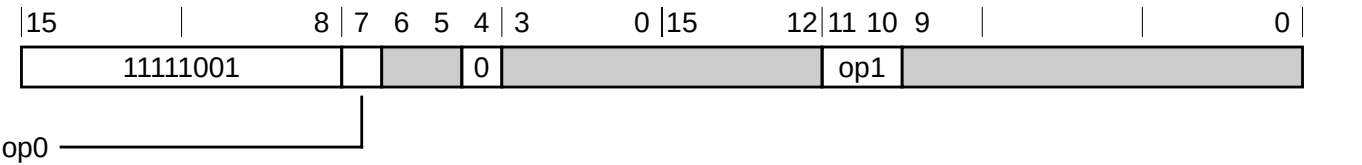
- hw1[15:13] of the T32 encoding is equivalent to bits[27:25] of the A32 encoding, and:
 - Has the value 0b111 in the T32 encoding.
 - Has the value 0b001 in the A32 encoding.
- hw1[11:8] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and has the value 0b111.

This table shows the equivalence of the fields that identify the instructions, or instruction encodings, within this group:

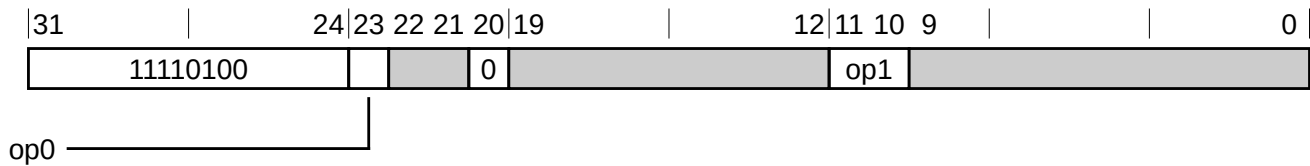
T32 encoding	A32 encoding	Field size
op0:op1	op0	2 bits
op2	op1	15 bits
op3	op2	1 bit
op4	op3	1 bit

C1.7.4.2 Advanced SIMD element or structure load/store

The T32 encoding of the Advanced SIMD element or structure load/store group is:



The A32 encoding of the Advanced SIMD element or structure load/store group is:



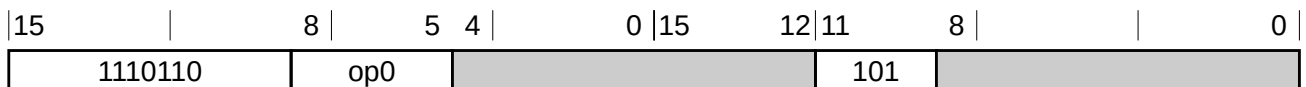
The encodings in this group are identified by:

- hw1[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and has the value 0b1111.
- hw1[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and:
 - Has the value 0b1001 in the T32 encoding.
 - Has the value 0b0100 in the A32 encoding.
- hw1[4] of the T32 encoding is equivalent to bit[20] of the A32 encoding, and has the value 0b0.

op0, op1, and op2 are the fields that identify the instructions, or instruction encodings, within this group, and they are in equivalent positions in the T32 and A32 encodings.

C1.7.4.3 Advanced SIMD and floating-point load/store and 64-bit register moves

The T32 encoding of the Advanced SIMD and floating-point load/store and 64-bit register moves group is:



The A32 encoding of the Advanced SIMD and floating-point load/store and 64-bit register moves group is:



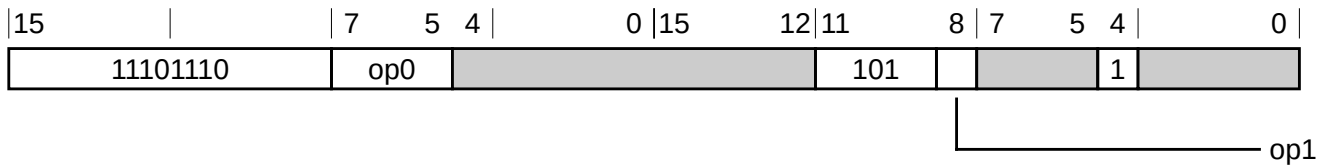
The encodings in the group are identified by:

- hw1[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
 - Has the value 0b1110 in the T32 encoding.
 - Can have any value other than 0b1111 in the A32 encoding.
This range of values is required because A32 instructions in this group can be executed conditionally, see [Conditional execution](#).
- hw1[11:9] of the T32 encoding is equivalent to bits[27:25] of the A32 encoding, and has the value 0b110.
- hw2[11:9] of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value 0b101.

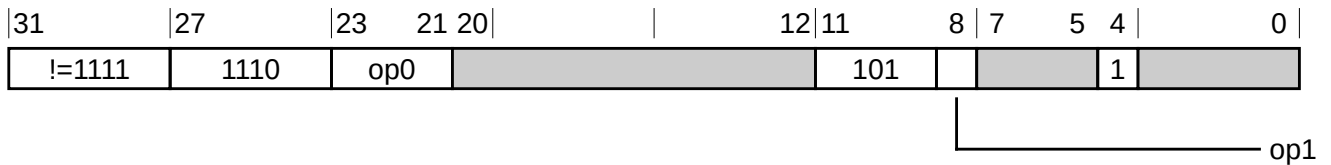
op0 is the field that identifies the instructions, or instruction encodings, within this group, and is in equivalent positions in the T32 and A32 encodings.

C1.7.4.4 Advanced SIMD and floating-point 32-bit register moves

The T32 encoding of the Advanced SIMD and floating-point 32-bit register moves group is:



The A32 encoding of the Advanced SIMD 32-bit register moves group is:



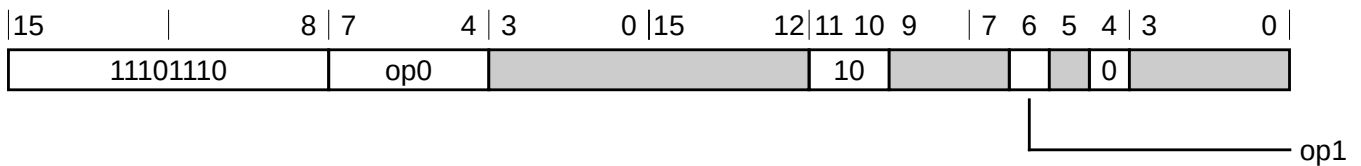
The encodings in this group are identified by:

- hw1[15:12] of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
 - Has the value 0b1110 in the T32 encoding.
 - Can have any value other than 0b1111 in the A32 encoding.
This range of values is required because A32 instructions in this group can be executed conditionally, see [Conditional execution](#).
- hw1[11:8] of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and has the value 0b1110.
- hw2[11:9] of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value 0b101.
- hw2[4] of the T32 encoding is equivalent to bit[4] of the A32 encoding, and has the value 0b1.

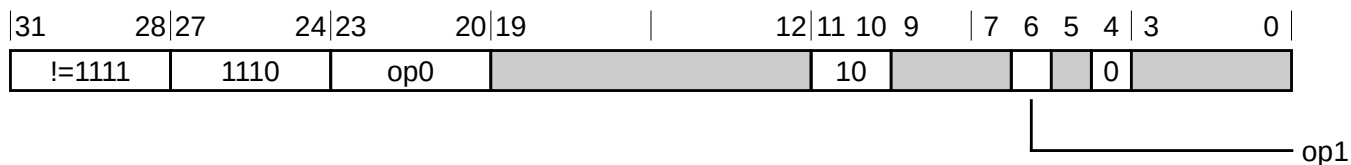
op0 is the field that identifies the instructions, or instruction encodings, within this group, and is in equivalent positions in the T32 and A32 encodings.

C1.7.4.5 Floating-point data-processing

The T32 encoding of the Floating-point data-processing group is:



The A32 encoding of the Floating-point data-processing group is:



The encodings in this group are identified by:

- `hw1[15:12]` of the T32 encoding is equivalent to bits[31:28] of the A32 encoding, and:
 - In the T32 encoding, `hw1[15:13]` has the value `0b1111`, and `hw1[12]` is the `op0` parameter used in identifying instruction encodings within this group.
 - In the A32 encoding, is the `cond` field and also implies the value of bit[28] of some A32 instruction encodings within this group, as the following table shows:

cond	Significance of bit[28] in A32 encodings
<code>!= 0b1111</code>	Part of the <code>cond</code> field.
<code>0b1111</code>	Has fixed value of 1.

The range of `cond` values other than `0b1111` is required because A32 instructions in this group can be executed conditionally, see [Conditional execution](#).

- `hw1[11:8]` of the T32 encoding is equivalent to bits[27:24] of the A32 encoding, and has the value `0b1110`.
- `hw2[11:9]` of the T32 encoding is equivalent to bits[11:9] of the A32 encoding, and has the value `0b101`.
- `hw2[4]` of the T32 encoding is equivalent to bit[4] of the A32 encoding, and has the value `0b0`.

This table shows the equivalence of the fields that identify the instructions, or instruction encodings, within this group:

T32 encoding	A32 encoding
<code>op0</code>	Bit[28] of the instruction encoding is 1 when <code>cond</code> is <code>0b1111</code> .
<code>op1</code>	<code>op0</code>
<code>op2</code>	<code>op1</code>
<code>op3</code>	<code>op2</code>

C1.7.5 The PC and the use of `0b1111` as a register specifier in T32 and A32 instructions

Restrictions on the use of PC or `0b1111` as a register specifier differ between the T32 and the A32 instruction sets, as described in:

- [T32 restrictions on the use of the PC, and use of `0b1111` as a register specifier.](#)
- [A32 restrictions on the use of PC or `0b1111` as a register specifier.](#)

C1.7.5.1 T32 restrictions on the use of the PC, and use of 0b1111 as a register specifier

The use of 0b1111 as a register specifier is not normally permitted in T32 instructions. When a value of 0b1111 is permitted, various meanings are possible. For register reads, these meanings include:

- Read the PC value, that is, the address of the current instruction + 4. The base register of the table branch instructions *TBB* and *TBH* can be the PC. This means branch tables can be placed in memory immediately after the instruction.

Note

Arm deprecates use of the PC as the base register in the *STC* instruction.

- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits[1:0] forced to zero. The base register of *LDC*, *LDR*, *LDRB*, *LDRD* (pre-indexed, no writeback), *LDRH*, *LDRSB*, and *LDRSH* instructions can be the word-aligned PC. This provides PC-relative data addressing. In addition, some encodings of the *ADD* and *SUB* instructions permit their source registers to be 0b1111 for the same purpose.
- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

For register writes, these meanings include:

- The PC can be specified as the destination register of an *LDR* instruction. This is done by encoding *Rt* as 0b1111. The loaded value is treated as an address, and the effect of execution is a branch to that address. Bit[0] of the loaded value selects whether to execute A32 or T32 instructions after the branch.
- Some other instructions write the PC in similar ways. An instruction can specify that the PC is written:
 - Implicitly, for example, branch instructions.
 - Explicitly by a register specifier of 0b1111, for example 16-bit *MOV* (register) instructions.
 - Explicitly by using a register mask, for example *LDM* instructions.

The address to branch to can be:

- A loaded value, for example, *RFE*.
- A register value, for example, *BX*.
- The result of a calculation, for example, *TBB* or *TBH*.

The method of choosing the instruction set used after the branch can be:

- Similar to the *LDR* case, for example, *LDM* or *BX*.
- A fixed instruction set other than the one currently being used, for example, the immediate form of *BLX*.
- Unchanged, for example, branch instructions or 16-bit *MOV* (register) instructions.
- Set from the *SPSR.T* bit, for *RFE* and *SUBS PC, LR, #imm8*.

- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.
- If the destination register specifier of an *LDRB*, *LDRH*, *LDRSB*, or *LDRSH* instruction is 0b1111, the instruction is a memory hint instead of a load operation.
- If the destination register specifier of an *MRC* instruction is 0b1111, bits[31:28] of the value transferred from the System register are written to the *N*, *Z*, *C*, and *V* condition flags in the *APSR*, and bits[27:0] are discarded.

C1.7.5.2 A32 restrictions on the use of PC or 0b1111 as a register specifier

In A32 instructions, the use of 0b1111 as a register specifier specifies the PC.

Many instructions are *CONSTRAINED UNPREDICTABLE* if they use 0b1111 as a register specifier. This is specified by pseudocode in the instruction description. Armv8-R constrains the resulting *CONSTRAINED UNPREDICTABLE* behavior, see [Using R15 by instruction](#).

Note

Arm deprecates use of the PC as the base register in any store instruction.

C1.7.6 The SP and the use of 0b1101 as a register specifier in T32 and A32 instructions

In the T32 and A32 instruction sets, Arm recommends that the use of 0b1101 as a register specifier specifies the SP.

Note

- The recommendation that the register specifier 0b1101 is used only to specify the SP applies to both the T32 and the A32 instruction sets.
- Despite this recommendation, T32 instructions that can access R13, or the SP, behave predictably from the introduction of Armv8.

C1.7.7 Modified immediate constants in T32 and A32 instructions

The following sections describe the encoding of modified immediate constants:

- [Modified immediate constants in T32 instructions.](#)
- [Modified immediate constants in A32 instructions.](#)
- [Modified immediate constants in T32 and A32 Advanced SIMD instructions.](#)
- [Modified immediate constants in T32 and A32 floating-point instructions.](#)

C1.7.7.1 Modified immediate constants in T32 instructions

The encoding of a modified immediate constant in a 32-bit T32 instruction is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					i												imm3							a	b	c	d	e	f	g	h

[Table C1-5](#) shows the range of modified immediate constants available in T32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, h, and i bits, and the imm3 field, in the instruction.

Table C1-5 Encoding of modified immediates in T32 data-processing instructions

i:imm3:a	<const> ^a
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh ^b
0010x	abcdefgh 00000000 abcdefgh 00000000 ^b
0011x	abcdefgh abcdefgh abcdefgh abcdefgh ^b
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000 ^c
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000 ^c
.	.
.	.
.	.
11101	00000000 00000000 000001bc defgh000 ^c
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0 ^c

- a. This table shows the immediate constant value in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).
- b. Arm deprecates using a modified immediate with `abcdefgh == 00000000`, and these cases are **CONSTRAINED UNPREDICTABLE**, see [UNPREDICTABLE cases in immediate constants in T32 data-processing instructions](#).
- c. Not available in A32 instructions if `h == 1`.

Note

As the footnotes to [Table C1-5](#) show, the range of values available in T32 modified immediate constants is slightly different from the range of values available in A32 instructions. See [Modified immediate constants in A32 instructions](#) for the A32 values.

C1.7.7.1.1 Carry out

A logical instruction with `i::imm3::a == '00xxx'` does not affect the Carry flag. Otherwise, a logical flag-setting instruction sets the Carry flag to the value of `bit[31]` of the modified immediate constant.

C1.7.7.1.2 Operation of modified immediate constants, T32 instructions

For a T32 data-processing instruction, the `T32ExpandImm()` pseudocode function returns the value of the 32-bit immediate constant, calling `T32ExpandImm_C()` to evaluate the constant.

C1.7.7.2 Modified immediate constants in A32 instructions

The encoding of a modified immediate constant in an A32 instruction is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																					rotation		a	b	c	d	e	f	g	h	

Table C1-6 shows the range of modified immediate constants available in A32 data-processing instructions, and their encoding in the a, b, c, d, e, f, g, and h bits and the rotation field in the instruction.

Table C1-6 Encoding of modified immediates in A32 processing instructions

rotation	<const> ^a
0000	00000000 00000000 00000000 abcdefgh
0001	gh000000 00000000 00000000 00abcdef
0010	efgh0000 00000000 00000000 0000abcd
0011	cdefgh00 00000000 00000000 000000ab
0100	abcdefgh 00000000 00000000 00000000
.	.
.	.
.	.
1001	00000000 00abcdef gh000000 00000000
.	.
.	.
.	.
1110	00000000 00000000 0000abcd efgh0000
1111	00000000 00000000 000000ab cdefgh00

a. This table shows the immediate constant value in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

Note

The range of values available in A32 modified immediate constants is slightly different from the range of values available in 32-bit T32 instructions. See [Modified immediate constants in T32 instructions](#).

C1.7.7.2.1 Carry out

A logical instruction with the rotation field set to 0b0000 does not affect APSR.C. Otherwise, a logical flag-setting instruction sets APSR.C to the value of bit[31] of the modified immediate constant.

C1.7.7.2.2 Constants with multiple encodings

Some constant values have multiple possible encodings. In this case, a UAL assembler must select the encoding with the lowest unsigned value of the rotation field. This is the encoding that appears first in Table C1-6. For example, the constant #3 must be encoded with (rotation, abcdefgh) = (0b0000, 0b00000011), not (0b0001, 0b00001100), (0b0010, 0b00110000), or (0b0011, 0b11000000).

In particular, this means that all constants in the range 0-255 are encoded with rotation == 0b0000, and permitted constants outside that range are encoded with rotation != 0b0000. A flag-setting logical instruction with a modified immediate constant therefore leaves APSR.C unchanged if the constant is in the range 0-255 and sets it to the most significant bit of the constant otherwise. This matches the behavior of T32 modified immediate constants for all constants that are permitted in both the T32 and A32 instruction sets.

An alternative syntax is available for a modified immediate constant that permits the programmer to specify the encoding directly. In this syntax, #<const> is instead written as #<byte>, #<rot>, where:

<byte> Is the numeric value of abcdefgh, in the range 0-255.

<rot> Is twice the numeric value of rotation, an even number in the range 0-30.

This syntax permits all A32 data-processing instructions with modified immediate constants to be disassembled to assembler syntax that assembles to the original instruction.

This syntax also makes it possible to write variants of some flag-setting logical instructions that have different effects on APSR.C to those obtained with the normal #<const> syntax. For example, `ANDS R1, R2, #12, #2` has the same behavior as `ANDS R1, R2, #3` except that it sets APSR.C to 0 instead of leaving it unchanged. Such variants of flag-setting logical instructions do not have equivalents in the T32 instruction set, and Arm deprecates their use.

C1.7.7.2.3 Operation of modified immediate constants, A32 instructions

For an A32 data-processing instruction, the `A32ExpandImm()` pseudocode function returns the value of the 32-bit immediate constant, calling `A32ExpandImm_C()` to evaluate the constant.

C1.7.7.3 Modified immediate constants in T32 and A32 Advanced SIMD instructions

Table C1-7 shows the modified immediate constants available with Advanced SIMD instructions, and how they are encoded.

Table C1-7 Modified immediate values for Advanced SIMD instructions

op	cmode	Constant ^a	<dt> ^b	Notes
-	000x	00000000 00000000 00000000 abcdefgh 00000000 00000000 00000000	I32	^c
		abcdefgh		
	001x	00000000 00000000 abcdefgh 00000000 00000000 00000000 abcdefgh	I32	^{c,d}
		00000000		
	010x	00000000 abcdefgh 00000000 00000000 00000000 abcdefgh 00000000	I32	^{c,d}
		00000000		
	011x	abcdefgh 00000000 00000000 00000000 abcdefgh 00000000 00000000	I32	^{c,d}
		00000000		
0	100x	00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000	I16	^c
		abcdefgh		
	101x	abcdefgh 00000000 abcdefgh 00000000 abcdefgh 00000000 abcdefgh	I16	^{c,d}
		00000000		
	1100	00000000 00000000 abcdefgh 11111111 00000000 00000000 abcdefgh	I32	^{d,e}
		11111111		
	1101	00000000 abcdefgh 11111111 11111111 00000000 abcdefgh 11111111	I32	^{d,e}
		11111111		
0	1110	abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh abcdefgh	I8	^f
		abcdefgh		
	1111	aBbbbbbc defgh000 00000000 00000000 aBbbbbbc defgh000 00000000	F32	^{f,g}
		00000000		

op	cmode	Constant ^a	<dt> ^b	Notes
1	1110	aaaaaaaa bbbbbbbb cccccccc dddddddd eeeeeeee ffffffff gggggggg hhhhhhhh	I64	f
	1111	UNDEFINED	-	-

a. In this table, the immediate value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembler syntax, the constant is specified by a data type and a value of that type. That value is specified in the normal way (a decimal number by default) and is replicated enough times to fill the 64-bit immediate. For example, a data type of I32 and a value of 10 specify the 64-bit constant 0x0000000A0000000A.

b. This specifies the data type used when the instruction is disassembled. On assembly, the data type must be matched in the table if possible. Other data types are permitted as pseudo-instructions when a program is assembled, provided the 64-bit constant specified by the data type and value is available for the instruction. If a constant is available in more than one way, the first entry in this table that can produce it is used. For example, `VMOV.I64 D0, #0x8000000080000000` does not specify a 64-bit constant that is available from the I64 line of the table, but does specify one that is available from the fourth I32 line or the F32 line. It is assembled to the first of these, and therefore is disassembled as `VMOV.I32 D0, #0x80000000`.

c. This constant is available for the VBIC, VMOV, VMVN, and VORR instructions.

d. CONSTRAINED UNPREDICTABLE if `abcdefgh == 0b00000000`, see [UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions](#). The required behavior is that these encodings produce an immediate constant of zero.

e. This constant is available for the VMOV and VMVN instructions only.

f. This constant is available for the VMOV instruction only.

g. In this entry, $B = \text{NOT}(b)$. The bit pattern represents the floating-point number $(-1)^S \times 2^{\text{exp}} \times \text{mantissa}$, where $S = \text{UInt}(a)$, $\text{exp} = \text{UInt}(\text{NOT}(b) : c : d) - 3$ and $\text{mantissa} = (16 + \text{UInt}(e : f : g : h)) / 16$.

C1.7.7.3.1 Operation of modified immediate constants, Advanced SIMD instructions

For a T32 or A32 Advanced SIMD instruction that uses a modified immediate constant, the operation described by the [AdvSIMDExpandImm\(\)](#) pseudocode function returns the value of the 64-bit immediate constant.

C1.7.7.4 Modified immediate constants in T32 and A32 floating-point instructions

[Table C1-8](#) shows the immediate constants available in the `VMOV` (immediate) floating-point instruction, and [Table C1-9](#) shows the resulting floating-point values.

Table C1-8 Floating-point modified immediate constants

Data type	imm4H	imm4L	Constant ^a
F16	abcd	efgh	aBbbcd ef gh000000
F32	abcd	efgh	aBbbbbbc defgh000 00000000 00000000
F64	abcd	efgh	aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000

a. In this column, B = NOT(b). The bit pattern represents the floating-point number $(-1)^S \times 2^{\text{exp}} \times \text{mantissa}$, where $S = \text{UInt}(a)$, $\text{exp} = \text{UInt}(\text{NOT}(b) : c : d) - 3$ and $\text{mantissa} = (\text{UInt}(e : f : g : h)) / 16$.

Table C1-9 Floating-point constant values

efgh	bcd							
	000	001	010	011	100	101	110	111
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

C1.7.7.4.1 Operation of modified immediate constants, floating-point instructions

For a T32 or A32 floating-point instruction that uses a modified immediate constant, the operation described by the [VFPEExpandImm\(\)](#) pseudocode function returns the value of the immediate constant.

C1.8 Additional pseudocode support for instruction descriptions

Earlier sections of this chapter include pseudocode that describes features of the execution of T32 and A32 instructions, see:

- [Pseudocode description of conditional execution.](#)
- [Pseudocode description of instruction-specified shifts and rotates](#)

The following subsection gives additional pseudocode support functions for some of the instructions described in [Alphabetical list of T32 and A32 base instruction set instructions](#). See also [Pseudocode support for the banked register transfer instructions](#).

C1.8.1 Pseudocode description of operations for System register access instructions

The `AArch32_SysRegRead()` function obtains the word for an MRC instruction from the System register.

The `AArch32_SysRegRead64()` function obtains the two words for an MRRC instruction from the System register.

Note

The relative significance of the two words returned is IMPLEMENTATION DEFINED, but all uses within this manual present the two words in the order (most significant, least significant).

The `AArch32_SysRegWrite()` procedure sends the word for an MCR instruction to the System register.

The `AArch32_SysRegWrite64()` procedure sends the two words for an MCRR instruction to the System register.

Note

The relative significance of `word2` and `word1` is IMPLEMENTATION DEFINED, but all uses within this manual treat `word2` as more significant than `word1`.

C1.8.2 Pseudocode details of system calls

The `AArch32_CallSupervisor()` pseudocode function generates a Supervisor Call exception. Valid execution of the SVC instruction calls this function.

The `AArch32_CallHypervisor()` pseudocode function generates an HVC exception. Valid execution of the HVC instruction calls this function.

C1.9 Additional information about Advanced SIMD and floating-point instructions

The following subsections give additional information about the Advanced SIMD and floating-point instructions:

- [Advanced SIMD and floating-point instruction syntax.](#)
- [The Advanced SIMD addressing mode.](#)
- [Advanced SIMD instruction modifiers.](#)
- [Advanced SIMD operand shapes.](#)
- [Data type specifiers.](#)
- [Register specifiers.](#)
- [Register lists.](#)
- [Register encoding.](#)
- [Advanced SIMD scalars.](#)

Note

The Advanced SIMD architecture, its associated implementations, and supporting software, are commonly referred to as NEON™ technology.

C1.9.1 Advanced SIMD and floating-point instruction syntax

Advanced SIMD and floating-point instructions use the general conventions of the T32 and A32 instruction sets.

Advanced SIMD and floating-point data-processing instructions use the following general format:

```
V{<modifier>}<operation>{<shape>}{<c>}{<q>}{<dt>} {<dest>}, {<src1>}, {<src2>}
```

All Advanced SIMD and floating-point instructions begin with a `v`. This distinguishes Advanced SIMD vector and floating-point instructions from scalar instructions.

The main operation is specified in the `<operation>` field. It is usually a three letter mnemonic the same as or similar to the corresponding scalar integer instruction.

The `<c>` and `<q>` fields are standard assembler syntax fields. For details, see [Standard assembler syntax fields](#).

C1.9.2 The Advanced SIMD addressing mode

All the element and structure load/store instructions use this addressing mode. There is a choice of three formats:

`[<Rn>{:<align>}]` The address is contained in general-purpose register `Rn`.

`Rn` is not updated by this instruction.

Encoded as `Rm = 0b1111`.

If `Rn` is encoded as `0b1111`, the instruction is CONSTRAINED UNPREDICTABLE.

`[<Rn>{:<align>}]!` The address is contained in general-purpose register `Rn`.

`Rn` is updated by this instruction: `Rn = Rn + transfer_size`

Encoded as `Rm = 0b1101`.

`transfer_size` is the number of bytes transferred by the instruction. This means that, after the instruction is executed, `Rn` points to the address in memory immediately following the last address loaded from or stored to.

If `Rn` is encoded as `0b1111`, the instruction is CONSTRAINED UNPREDICTABLE.

This addressing mode can also be written as:

`[<Rn>{:<align>}], #<transfer_size>`

However, disassembly produces the `[<Rn>{:<align>}]!` form.

[<Rn>{:<align>}], <Rm>

The address is contained in general-purpose register <Rn>.

Rn is updated by this instruction: $Rn = Rn + Rm$

Encoded as Rm = Rm. Rm must not be encoded as 0b1111 or 0b1101, the PC or the SP.

If Rn is encoded as 0b1111, the instruction is CONSTRAINED UNPREDICTABLE.

The CONSTRAINED UNPREDICTABLE behavior of encodings where Rn is 0b1111 is described in the section: [Using R15 by instruction](#).

In all cases, <align> specifies an alignment, as specified by the individual instruction descriptions.

Previous versions of the manual used the @ character for alignment. So, for example, the first format in this section was shown as [<Rn>{@<align>}]. Both @ and : are supported. However, to ensure portability of code to assemblers that treat @ as a comment character, : is preferred.

C1.9.3 Advanced SIMD instruction modifiers

The <modifier> field provides additional variants of some instructions. [Table C1-10](#) provides definitions of the modifiers. Modifiers are not available for every instruction.

Table C1-10 Advanced SIMD instruction modifiers

<modifier>	Meaning
Q	The operation uses saturating arithmetic.
R	The operation performs rounding.
D	The operation doubles the result (before accumulation, if any).
H	The operation halves the result.

C1.9.4 Advanced SIMD operand shapes

The <shape> field provides additional variants of some instructions. [Table C1-11](#) provides definitions of the shapes. Operand shapes are not available for every instruction.

Table C1-11 Advanced SIMD operand shapes

<shape>	Meaning	Typical register shape
(none)	The operands and result are all the same width.	Dd, Dn, Dm Qd, Qn, Qm
L	Long operation - result is twice the width of both operands	Qd, Dn, Dm
N	Narrow operation - result is half the width of both operands	Dd, Qn, Qm
W	Wide operation - result and first operand are twice the width of the second operand	Qd, Qn, Dm

Note

- Some assemblers support a Q shape specifier, which requires all operands to be Q registers. An example of using this specifier is `VADDQ.S32 q0, q1, q2`. This is not standard UAL, and Arm recommends that programmers do not use a Q shape specifier.
- A disassembler must not generate any shape specifier not shown in [Table C1-11](#).

C1.9.5 Data type specifiers

The <dt> field normally contains one data type specifier. Unless the assembler syntax description for the instruction indicates otherwise, this indicates the data type contained in:

- The second operand, if any.
- The operand, if there is no second operand.
- The result, if there are no operand registers.

The data types of the other operand and result are implied by the <dt> field combined with the instruction shape. For information about data type formats, see [Data types supported by the Advanced SIMD implementation](#).

In the instruction syntax descriptions in [About the T32 and A32 Instruction Descriptions](#), the <dt> field is usually specified as a single field. However, where more convenient, it is sometimes specified as a concatenation of two fields, <type><size>.

C1.9.5.1 Syntax flexibility

There is some flexibility in the data type specifier syntax:

- Software can specify three data types, specifying the result and both operand data types. For example:
VSUBW.I16.I16.S8 Q3, Q5, D0 instead of VSUBW.S8 Q3, Q5, D0
- Software can specify two data types, specifying the data types of the two operands. The data type of the result is implied by the instruction shape. For example:
VSUBW.I16.S8 Q3, Q5, D0 instead of VSUBW.S8 Q3, Q5, D0
- Software can specify two data types, specifying the data types of the single operand and the result. For example:
VMOVN.I16.I32 D0, Q1 instead of VMOVN.I32 D0, Q1
- Where an instruction requires a less specific data type, software can instead specify a more specific type, as shown in [Table C1-12](#).
- Where an instruction does not require a data type, software can provide one.
- The F32 data type can be abbreviated to F.
- The F64 data type can be abbreviated to D.

In all cases, if software provides additional information, the additional information must match the instruction shape. Disassembly does not regenerate this additional information.

Table C1-12 Data type specification flexibility

Specified data type	Permitted more specific data types				
None	Any				
.I<size>	-	.S<size>	.U<size>	-	-
.8	.I8	.S8	.U8	.P8	-
.16	.I16	.S16	.U16	.P16	.F16
.32	.I32	.S32	.U32	-	.F32 or .F
.64	.I64	.S64	.U64	-	.F64 or .D

C1.9.6 Register specifiers

The `<dest>`, `<src1>`, and `<src2>` fields contain register specifiers, or in some cases scalar specifiers or register lists. [Table C1-13](#) shows the register and scalar specifier formats that appear in the instruction descriptions.

If `<dest>` is omitted, it is the same as `<src1>`.

Table C1-13 Advanced SIMD and floating-point register specifier formats

<specifier>	Usual meaning^a	Used in
<code><Qd></code>	A quadword destination register for the result vector.	Advanced SIMD
<code><Qn></code>	A quadword source register for the first operand vector.	Advanced SIMD
<code><Qm></code>	A quadword source register for the second operand vector.	Advanced SIMD
<code><Dd></code>	A doubleword destination register for the result vector.	Both
<code><Dn></code>	A doubleword source register for the first operand vector.	Both
<code><Dm></code>	A doubleword source register for the second operand vector.	Both
<code><Sd></code>	A singleword destination register for the result vector.	Floating-point
<code><Sn></code>	A singleword source register for the first operand vector.	Floating-point
<code><Sm></code>	A singleword source register for the second operand vector.	Floating-point
<code><Dd[x]></code>	A destination scalar for the result. Element x of vector <code><Dd></code> .	Advanced SIMD
<code><Dn[x]></code>	A source scalar for the first operand. Element x of vector <code><Dn></code> .	Both ^b
<code><Dm[x]></code>	A source scalar for the second operand. Element x of vector <code><Dm></code> .	Advanced SIMD
<code><Rt></code>	A general-purpose register, used for a source or destination address.	Both
<code><Rt2></code>	A general-purpose register, used for a source or destination address.	Both
<code><Rn></code>	A general-purpose register, used as a load or store base address.	Both
<code><Rm></code>	A general-purpose register, used as a post-indexed address source.	Both

a. In some instructions the roles of registers are different.

b. In the floating-point instructions, `<Dn[x]>` is used only in `VMOV` (scalar to general-purpose register), see [VMOV \(scalar to general-purpose register\)](#).

C1.9.7 Register lists

A register list is a list of register specifiers separated by commas and enclosed in brackets { and }. There are restrictions on what registers can appear in a register list. These restrictions are described in the individual instruction descriptions. [Table C1-14](#) shows some register list formats, with examples of actual register lists corresponding to those formats.

Note

Register lists must not wrap around the end of the register bank.

C1.9.7.1 Syntax flexibility

There is some flexibility in the register list syntax:

- Where a register list contains consecutive registers, they can be specified as a range, instead of listing every register, for example {D0–D3} instead of {D0, D1, D2, D3}.
- Where a register list contains an even number of consecutive doubleword registers starting with an even-numbered register, it can be written as a list of quadword registers instead, for example {Q1, Q2} instead of {D2–D5}.
- Where a register list contains only one register, the enclosing braces can be omitted, for example VLD1.8 D0, [R0] instead of VLD1.8 {D0}, [R0].

Table C1-14 Example register lists

Format	Example	Alternative
{<Dd>}	{D3}	D3
{<Dd>, <Dd+1>, <Dd+2>}	{D3, D4, D5}	{D3–D5}
{<Dd[x]>, <Dd+2[x]>}	{D0[3], D2[3]}	-
{<Dd[]>}	{D7[]}	D7[]

C1.9.8 Register encoding

An Advanced SIMD register is either:

- Quadword*, meaning it is 128 bits wide.
- Doubleword*, meaning it is 64 bits wide.

Some instructions have options for either doubleword or quadword registers. This is normally encoded in Q, bit[6], as Q = 0 for doubleword operations, or Q = 1 for quadword operations.

A floating-point register is either:

- Double-precision, meaning it is 64 bits wide.
- Single-precision, meaning it is 32 bits wide.

This is encoded in the sz field, bit[8], as sz = 1 for double-precision operations, or sz = 0 for single-precision operations.

The T32 instruction encoding of Advanced SIMD or floating-point registers is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
									D				Vn						Vd						sz	N	Q	M				Vm		

The A32 instruction encoding of Advanced SIMD or floating-point registers is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
									D				Vn			Vd						sz	N	Q	M				Vm		

Some instructions use only one or two registers, and use the unused register fields as additional opcode bits.

Table C1-15 shows the encodings for the registers.

Table C1-15 Encoding of register numbers

Register mnemonic	Usual usage	Register number encoded in ^a	Notes ^a	Used in
<Qd>	Destination (quadword)	D, Vd (bits[22, 15:13])	bit[12] == 0 ^b	Advanced SIMD
<Qn>	First operand (quadword)	N, Vn (bits[7, 19:17])	bit[16] == 0 ^b	Advanced SIMD
<Qm>	Second operand (quadword)	M, Vm (bits[5, 3:1])	bit[0] == 0 ^b	Advanced SIMD
<Dd>	Destination (doubleword)	D, Vd (bits[22, 15:12])	-	Both
<Dn>	First operand (doubleword)	N, Vn (bits[7, 19:16])	-	Both
<Dm>	Second operand (doubleword)	M, Vm (bits[5, 3:0])	-	Both
<Sd>	Destination (single-precision)	Vd, D (bits[15:12, 22])	-	Floating-point
<Sn>	First operand (single-precision)	Vn, N (bits[19:16, 7])	-	Floating-point
<Sm>	Second operand (single-precision)	Vm, M (bits[3:0, 5])	-	Floating-point

a. Bit numbers given for the A32 instruction encoding. See the figures in this section for the equivalent bits in the T32 encoding.

b. If this bit is 1, the instruction is UNDEFINED.

C1.9.9 Advanced SIMD scalars

Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. Instructions other than multiply instructions can access any element in the register set. The instruction syntax refers to the scalars using an index into a doubleword vector. The descriptions of the individual instructions contain details of the encodings.

Table C1-16 shows the form of encoding for scalars used in multiply instructions. These instructions cannot access scalars in some registers. The descriptions of the individual instructions contain cross references to this section where appropriate.

32-bit Advanced SIMD scalars, when used as single-precision floating-point numbers, are equivalent to Floating-point single-precision registers. That is, $D_m[x]$ in a 32-bit context ($0 \leq m \leq 15, 0 \leq x \leq 1$) is equivalent to $S[2m + x]$.

Table C1-16 Encoding of scalars in multiply instructions

Scalar mnemonic	Usual usage	Scalar size	Register specifier	Index specifier	Accessible registers
<Dm [x]>	Second operand	16-bit	Vm[2:0]	M, Vm[3]	D0-D7
		32-bit	Vm[3:0]	M	D0-D15

Chapter C2

The AArch32 Instruction Sets Overview

This chapter describes the T32 and A32 instruction sets. It contains the following sections:

- [Support for instructions in different versions of the Arm architecture](#)
- [Unified Assembler Language](#)
- [Branch instructions](#)
- [Data-processing instructions](#)
- [PSTATE and banked register access instructions](#)
- [Load/store instructions](#)
- [Load/store multiple instructions](#)
- [Miscellaneous instructions](#)
- [Exception-generating and exception-handling instructions](#)
- [System register access instructions](#)
- [Advanced SIMD and floating-point load/store instructions](#)
- [Advanced SIMD and floating-point register transfer instructions](#)
- [Advanced SIMD data-processing instructions](#)
- [Floating-point data-processing instructions](#)

C2.1 Support for instructions in different versions of the Arm architecture

This manual describes the T32 and A32 instruction sets for the architecture. Therefore, it indicates how any options or extensions in the architecture affect the available instructions.

C2.2 Unified Assembler Language

This manual uses the Arm *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all T32 and A32 instructions.

UAL describes the syntax for the mnemonic and the operands of each instruction. In addition, it assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, nor what assembler directives and options are available. See your assembler documentation for these details.

Most earlier Arm assembly language mnemonics are still supported as synonyms, as described in the instruction details.

Note

Most earlier T32 assembly language mnemonics are *not* supported.

UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality. For example, both 16-bit and 32-bit encodings exist for an `ADD R0, R1, R2` instruction. The most common instruction selection rule is that when both a 16-bit encoding and a 32-bit encoding are available, the 16-bit encoding is selected, to optimize code density.

Syntax options exist to override the normal instruction selection rules and ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

C2.2.1 Conditional instructions

For maximum portability of UAL assembly language between the T32 and A32 instruction sets, Arm recommends that:

- `IT` instructions are written before conditional instructions in the correct way for the T32 instruction set.
- When assembling to the A32 instruction set, assemblers check that any `IT` instructions are correct, but do not generate any code for them.

Although other T32 instructions are unconditional, all instructions that are made conditional by an `IT` instruction must be written with a condition. These conditions must match the conditions imposed by the `IT` instruction. For example, an `ITTEE EQ` instruction imposes the `EQ` condition on the first two following instructions, and the `NE` condition on the next two. Those four instructions must be written with `EQ`, `EQ`, `NE` and `NE` conditions respectively.

Some instructions cannot be made conditional by an `IT` instruction. Some instructions can be conditional if they are the last instruction in the `IT` block, but not otherwise.

The branch instruction encodings that include a condition code field cannot be made conditional by an `IT` instruction. If the assembler syntax indicates a conditional branch that correctly matches a preceding `IT` instruction, it is assembled using a branch instruction encoding that does not include a condition code field.

C2.2.2 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the `PC` or `Align(PC, 4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a T32 instruction, or plus 8 for an A32 instruction. The `Align(PC, 4)` value of an instruction is its `PC` value ANDed with `0xFFFFFEC` to force it to be word-aligned. There is no difference between the `PC` and `Align(PC, 4)` values for an A32 instruction, but there can be for a T32 instruction.
2. Calculate the offset from the `PC` or `Align(PC, 4)` value of the instruction to the address of the labeled instruction or literal data item.
3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its `PC` or `Align(PC, 4)` value and adds the calculated offset to form the required address.

Note

For instructions that can encode a subtraction operation, if the instruction cannot encode the calculated offset but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

The syntax of the following instructions includes a label:

- **B, BL, and BLX (immediate).** The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a sign-extended immediate offset that is added to the PC value of the instruction to form the target address of the branch.
- **CBNZ and CBZ.** The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a zero-extended immediate offset that is added to the PC value of the instruction to form the target address of the branch. They do not support backward branches.
- **LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH, PLD, PLDW, PLI, and VLDR.** The normal assembler syntax of these load instructions can specify the label of a literal data item that is to be loaded. The encodings of these instructions specify a zero-extended immediate offset that is either added to or subtracted from the `Align(PC, 4)` value of the instruction to form the address of the data item. A few such encodings perform a fixed addition or a fixed subtraction and must only be used when that operation is required, but most contain a bit that specifies whether the offset is to be added or subtracted.

When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the `Align(PC, 4)` value of the instruction. Encodings that subtract 0 from the `Align(PC, 4)` value cannot be specified by the normal syntax.

There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by `[PC, #+/-<imm>]`, where:

`+/-` Is + or omitted to specify that the immediate offset is to be added to the `Align(PC, 4)` value, or - if it is to be subtracted.

`<imm>` Is the immediate offset.

This alternative syntax makes it possible to assemble the encodings that subtract 0 from the `Align(PC, 4)` value, and to disassemble them to a syntax that can be re-assembled correctly.

- **ADR.** The normal assembler syntax for this instruction can specify the label of an instruction or literal data item whose address is to be calculated. Its encoding specifies a zero-extended immediate offset that is either added to or subtracted from the `Align(PC, 4)` value of the instruction to form the address of the data item, and some opcode bits that determine whether it is an addition or subtraction.

When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the `Align(PC, 4)` value of the instruction. The encoding that subtracts 0 from the `Align(PC, 4)` value cannot be specified by the normal syntax.

There is an alternative syntax for this instruction that specifies the addition or subtraction and the immediate value explicitly, by writing them as additions `ADD <Rd>, PC, #<imm>` or subtractions `SUB <Rd>, PC, #<imm>`. This alternative syntax makes it possible to assemble the encoding that subtracts 0 from the `Align(PC, 4)` value, and to disassemble it to a syntax that can be re-assembled correctly.

Note

Arm recommends that where possible, software avoids using:

- The alternative syntax for the **ADR, LDC, LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH, PLD, PLI, PLDW, and VLDR** instructions.
 - The encodings of these instructions that subtract 0 from the `Align(PC, 4)` value.
-

C2.3 Branch instructions

[Table C2-1](#) summarizes the branch instructions in the T32 and A32 instruction sets. In addition to providing for changes in the flow of execution, some branch instructions can change instruction set.

Table C2-1 Branch instructions

Instruction	See	Range, T32	Range, A32
Branch to target address	B	±16MB	±32MB
Compare and Branch on Nonzero Compare and Branch on Zero	CBNZ, CBZ	0-126 bytes	-
Call a subroutine Call a subroutine, change instruction set ^a	BL, BLX (immediate)	±16MB ±16MB	±32MB ±32MB
Call a subroutine, optionally change instruction set	BLX (register)	Any	Any
Branch to target address, change instruction set	BX	Any	Any
Change to Jazelle state	BXJ	-	-
Table Branch (byte offsets) Table Branch (halfword offsets)	TBB, TBH	0-510 bytes 0-131070 bytes	-

a. The range is determined by the instruction set of the BLX instruction, not of the instruction it branches to.

Branches to loaded and calculated addresses can be performed by `LDR`, `LDM` and data-processing instructions. For details, see [Load/store instructions](#), [Load/store multiple instructions](#), [Standard data-processing instructions](#), and [Shift instructions](#).

In addition to the branch instructions shown in [Table C2-1](#):

- In the A32 instruction set, a data-processing instruction that targets the PC behaves as a branch instruction. For more information, see [Data-processing instructions](#).
- In the T32 and A32 instruction sets, a load instruction that targets the PC behaves as a branch instruction. For more information, see [Load/store instructions](#).

C2.4 Data-processing instructions

Core data-processing instructions belong to one of the following groups:

- [Standard data-processing instructions.](#)
- [Shift instructions.](#)
- [Multiply instructions.](#)
- [Saturating instructions.](#)
- [Saturating addition and subtraction instructions.](#)
- [Packing and unpacking instructions.](#)
- [Parallel addition and subtraction instructions.](#)
- [Divide instructions.](#)
- [Miscellaneous data-processing instructions.](#)

For related Advanced SIMD and floating-point instructions see [Advanced SIMD data-processing instructions](#) and [Floating-point data-processing instructions](#).

C2.4.1 Standard data-processing instructions

These instructions perform basic data-processing operations, and share a common format with some variations.

These instructions generally have a destination register Rd, a first operand register Rn, and a second operand. The second operand can be another register Rm, or an immediate constant.

If the second operand is an immediate constant, it can be:

- Encoded directly in the instruction.
- A *modified immediate constant* that uses 12 bits of the instruction to encode a range of constants. T32 and A32 instructions have slightly different ranges of modified immediate constants. For more information, see [Modified immediate constants in T32 instructions](#) and [Modified immediate constants in A32 instructions](#).

If the second operand is another register, it can optionally be shifted in any of the following ways:

LSL	Logical Shift Left by 1-31 bits.
LSR	Logical Shift Right by 1-32 bits.
ASR	Arithmetic Shift Right by 1-32 bits.
ROR	Rotate Right by 1-31 bits.
RRX	Rotate Right with Extend. For details, see Shift and rotate operations .

In T32 code, the amount to shift by is always a constant encoded in the instruction. In A32 code, the amount to shift by is either a constant encoded in the instruction, or the value of a register, Rs.

For instructions other than CMN, CMP, TEQ, and TST, the result of the data-processing operation is placed in the destination register. In the A32 instruction set, the destination register can be the PC, causing the result to be treated as a branch address. In the T32 instruction set, this is only permitted for some 16-bit forms of the ADD and MOV instructions.

These instructions can optionally set the Condition flags, according to the result of the operation. If they do not set the flags, existing flag settings from a previous instruction are preserved.

[Table C2-2](#) summarizes the main data-processing instructions in the T32 and A32 instruction sets. Generally, each of these instructions is described in three sections in [About the T32 and A32 Instruction Descriptions](#), one section for each of the following:

- INSTRUCTION (immediate) where the second operand is a modified immediate constant.
- INSTRUCTION (register) where the second operand is a register, or a register shifted by a constant.
- INSTRUCTION (register-shifted register) where the second operand is a register shifted by a value obtained from another register. These are only available in the A32 instruction set.

Table C2-2 Standard data-processing instructions

Instruction	Mnemonic	Notes
Add with Carry	ADC	-
Add	ADD	T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
Form PC-relative Address	ADR	First operand is the PC. Second operand is an immediate constant. T32 instruction set uses a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
Bitwise AND	AND	-
Bitwise Bit Clear	BIC	-
Compare Negative	CMN	Sets flags. Like ADD but with no destination register.
Compare	CMP	Sets flags. Like SUB but with no destination register.
Bitwise Exclusive OR	EOR	-
Copy operand to destination	MOV	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. For details, see Shift instructions . The T32 and A32 instruction sets permit use of a modified immediate constant or a zero-extended 16-bit immediate constant.
Bitwise NOT	MVN	Has only one operand, with the same options as the second operand in most of these instructions.
Bitwise OR NOT	ORN	Not available in the A32 instruction set.
Bitwise OR	ORR	-
Reverse Subtract	RSB	Subtracts first operand from second operand. This permits subtraction from constants and shifted registers.
Reverse Subtract with Carry	RSC	Not available in the T32 instruction set.
Subtract with Carry	SBC	-
Subtract	SUB	T32 instruction set permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
Test Equivalence	TEQ	Sets flags. Like EOR but with no destination register.
Test	TST	Sets flags. Like AND but with no destination register.

C2.4.2 Shift instructions

[Table C2-3](#) lists the shift instructions in the T32 and A32 instruction sets.

Table C2-3 Shift instructions

Instruction	See
Arithmetic Shift Right	ASR (immediate) ASR (register) ASRS (immediate) ASRS (register)

Instruction	See
Logical Shift Left	LSL (immediate) LSL (register) LSLS (immediate) LSLS (register)
Logical Shift Right	LSR (immediate) LSR (register) LSRS (immediate) LSRS (register)
Rotate Right	ROR (immediate) ROR (register) RORS (immediate) RORS (register)
Rotate Right with Extend	RRX RRXS

In the A32 instruction set only, the destination register of these instructions can be the PC, causing the result to be treated as an address to branch to.

C2.4.3 Multiply instructions

These instructions can operate on signed or unsigned quantities. In some types of operation, the results are the same whether the operands are signed or unsigned.

- [Table C2-4](#) summarizes the multiply instructions where there is no distinction between signed and unsigned quantities.
The least significant 32 bits of the result are used. More significant bits are discarded.
- [Table C2-5](#) summarizes the signed multiply instructions.
- [Table C2-6](#) summarizes the unsigned multiply instructions.

Table C2-4 General multiply instructions

Instruction	See	Operation (number of bits)
Multiply Accumulate	MLA , MLAS	$32 = 32 + 32 \times 32$
Multiply and Subtract	MLS	$32 = 32 - 32 \times 32$
Multiply	MUL , MULS	$32 = 32 \times 32$

Table C2-5 Signed multiply instructions

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate (halfwords)	SMLABB , SMLABT , SMLATB , SMLATT	$32 = 32 + 16 \times 16$
Signed Multiply Accumulate Dual	SMLAD , SMLADX	$32 = 32 + 16 \times 16 + 16 \times 16$
Signed Multiply Accumulate Long	SMLAL , SMLALS	$64 = 64 + 32 \times 32$

Instruction	See	Operation (number of bits)
Signed Multiply Accumulate Long (halfwords)	SMLALBB , SMLALBT , SMLALTB , SMLALTT	$64 = 64 + 16 \times 16$
Signed Multiply Accumulate Long Dual	SMLALD , SMLALDX	$64 = 64 + 16 \times 16 + 16 \times 16$
Signed Multiply Accumulate (word by halfword)	SMLAWB , SMLAWT	$32 = 32 + 32 \times 16^a$
Signed Multiply Subtract Dual	SMLSD , SMLSXD	$32 = 32 + 16 \times 16 - 16 \times 16$
Signed Multiply Subtract Long Dual	SMLSLD , SMLSLDX	$64 = 64 + 16 \times 16 - 16 \times 16$
Signed Most Significant Word Multiply Accumulate	SMMLA , SMMLAR	$32 = 32 + 32 \times 32^b$
Signed Most Significant Word Multiply Subtract	SMMLS , SMMLSR	$32 = 32 - 32 \times 32^b$
Signed Most Significant Word Multiply	SMMUL , SMMULR	$32 = 32 \times 32^b$
Signed Dual Multiply Add	SMUAD , SMUADX	$32 = 16 \times 16 + 16 \times 16$
Signed Multiply (halfwords)	SMULBB , SMULBT , SMULTB , SMULTT	$32 = 16 \times 16$
Signed Multiply Long	SMULL , SMULLS	$64 = 32 \times 32$
Signed Multiply (word by halfword)	SMULWB , SMULWT	$32 = 32 \times 16^a$
Signed Dual Multiply Subtract	SMUSD , SMUSDX	$32 = 16 \times 16 - 16 \times 16$

a. The most significant 32 bits of the 48-bit product are used. Less significant bits are discarded.

b. The most significant 32 bits of the 64-bit product are used. Less significant bits are discarded.

Table C2-6 Unsigned multiply instructions

Instruction	See	Operation (number of bits)
Unsigned Multiply Accumulate Accumulate Long	UMAAL	$64 = 32 + 32 + 32 \times 32$
Unsigned Multiply Accumulate Long	UMLAL , UMLALS	$64 = 64 + 32 \times 32$
Unsigned Multiply Long	UMULL , UMULLS	$64 = 32 \times 32$

C2.4.4 Saturating instructions

[Table C2-7](#) lists the saturating instructions in the T32 and A32 instruction sets. For more information, see [Pseudocode description of saturation](#).

Table C2-7 Saturating instructions

Instruction	See	Operation
Signed Saturate	SSAT	Saturates optionally shifted 32-bit value to selected range
Signed Saturate 16	SSAT16	Saturates two 16-bit values to selected range
Unsigned Saturate	USAT	Saturates optionally shifted 32-bit value to selected range
Unsigned Saturate 16	USAT16	Saturates two 16-bit values to selected range

C2.4.5 Saturating addition and subtraction instructions

[Table C2-8](#) lists the saturating addition and subtraction instructions in the T32 and A32 instruction sets. For more information, see [Pseudocode description of saturation](#).

Table C2-8 Saturating addition and subtraction instructions

Instruction	See	Operation
Saturating Add	QADD	Add, saturating result to the 32-bit signed integer range
Saturating Subtract	QSUB	Subtract, saturating result to the 32-bit signed integer range
Saturating Double and Add	QADD	Doubles one value and adds a second value, saturating the doubling and the addition to the 32-bit signed integer range
Saturating Double and Subtract	QDSUB	Doubles one value and subtracts the result from a second value, saturating the doubling and the subtraction to the 32-bit signed integer range

C2.4.6 Packing and unpacking instructions

[Table C2-9](#) lists the packing and unpacking instructions in the T32 and A32 instruction sets.

Table C2-9 Packing and unpacking instructions

Instruction	See	Operation
Pack Halfword	PKHBT , PKHTB	Combine halfwords
Signed Extend and Add Byte	SXTAB	Extend 8 bits to 32 and add
Signed Extend and Add Byte 16	SXTAB16	Dual extend 8 bits to 16 and add
Signed Extend and Add Halfword	SXTAH	Extend 16 bits to 32 and add
Signed Extend Byte	SXTB	Extend 8 bits to 32

Instruction	See	Operation
Signed Extend Byte 16	SXTB16	Dual extend 8 bits to 16
Signed Extend Halfword	SXTH	Extend 16 bits to 32
Unsigned Extend and Add Byte	UXTAB	Extend 8 bits to 32 and add
Unsigned Extend and Add Byte 16	UXTAB16	Dual extend 8 bits to 16 and add
Unsigned Extend and Add Halfword	UXTAH	Extend 16 bits to 32 and add
Unsigned Extend Byte	UXTB	Extend 8 bits to 32
Unsigned Extend Byte 16	UXTB16	Dual extend 8 bits to 16
Unsigned Extend Halfword	UXTH	Extend 16 bits to 32

C2.4.7 Parallel addition and subtraction instructions

These instructions perform additions and subtractions on the values of two registers and write the result to a destination register, treating the register values as sets of two halfwords or four bytes. That is, they perform SIMD additions or subtractions on the general-purpose registers.

These instructions consist of a prefix followed by a main instruction mnemonic. The prefixes are as follows:

S	Signed arithmetic modulo 2^8 or 2^{16} .
Q	Signed saturating arithmetic.
SH	Signed arithmetic, halving the results.
U	Unsigned arithmetic modulo 2^8 or 2^{16} .
UQ	Unsigned saturating arithmetic.
UH	Unsigned arithmetic, halving the results.

The main instruction mnemonics are as follows:

ADD16	Adds the top halfwords of two operands to form the top halfword of the result, and the bottom halfwords of the same two operands to form the bottom halfword of the result.
ASX	Exchanges halfwords of the second operand, and then adds top halfwords and subtracts bottom halfwords.
SAX	Exchanges halfwords of the second operand, and then subtracts top halfwords and adds bottom halfwords.
SUB16	Subtracts each halfword of the second operand from the corresponding halfword of the first operand to form the corresponding halfword of the result.
ADD8	Adds each byte of the second operand to the corresponding byte of the first operand to form the corresponding byte of the result.
SUB8	Subtracts each byte of the second operand from the corresponding byte of the first operand to form the corresponding byte of the result.

The instruction set permits all 36 combinations of prefix and main instruction operand, as [Table C2-10](#) shows.

See also [Advanced SIMD parallel addition and subtraction](#).

Table C2-10 Parallel addition and subtraction instructions

Main instruction	Signed	Saturating	Signed halving	Unsigned	Unsigned saturating	Unsigned halving
ADD16, add, two halfwords	SADD16	QADD16	SHADD16	UADD16	UQADD16	UHADD16
ASX, add and subtract with exchange	SASX	QASX	SHASX	UASX	UQASX	UHASX
SAX, subtract and add with exchange	SSAX	QSAX	SHSAX	USAX	UQSAX	UHSAX
SUB16, subtract, two halfwords	SSUB16	QSUB16	SHSUB16	USUB16	UQSUB16	UHSUB16
ADD8, add, four bytes	SADD8	QADD8	SHADD8	UADD8	UQADD8	UHADD8
SUB8, subtract, four bytes	SSUB8	QSUB8	SHSUB8	USUB8	UQSUB8	UHSUB8

C2.4.8 Divide instructions

Signed and unsigned integer divide instructions are included in both the T32 instruction set and the A32 instruction set.

For descriptions of the instructions see:

- SDIV.
- UDIV.

For the SDIV and UDIV

instructions, division by zero always returns a zero result.

The ID_ISAR0.Divide_instrs field indicates the level of support for these instructions. The field value of 0b0010 indicates they are implemented in both the T32 and A32 instruction sets.

C2.4.9 Miscellaneous data-processing instructions

Table C2-11 lists the miscellaneous data-processing instructions in the T32 and A32 instruction sets. Immediate values in these instructions are simple binary numbers.

Table C2-11 Miscellaneous data-processing instructions

Instruction	See	Notes
BitField Clear	BFC	-
BitField Insert	BFI	-
Count Leading Zeros	CLZ	-
Move Top	MOVT	Moves 16-bit immediate value to top halfword. Bottom halfword unchanged.
Reverse Bits	RBIT	-
Byte-Reverse Word	REV	-
Byte-Reverse Packed Halfword	REV16	-
Byte-Reverse Signed Halfword	REVSH	-
Signed BitField Extract	SBFX	-
Select Bytes using GE flags	SEL	-
Unsigned BitField Extract	UBFX	-
Unsigned Sum of Absolute Differences	USAD8	-
Unsigned Sum of Absolute Differences and Accumulate	USADA8	-

C2.5 PSTATE and banked register access instructions

These instructions transfer PE state information to or from a general-purpose register.

C2.5.1 PSTATE access instructions

PSTATE holds process state information, see [Process state, PSTATE](#). In AArch32 state:

- At EL1 or higher, [PSTATE](#) is accessible using the *Current Program Status Register (CPSR)*.
- At EL0, a subset of the [CPSR](#) is accessible as the *Application Program Status Register (APSR)*.
- On taking an exception, the contents of the [CPSR](#) are copied to the *Saved Program Status Register (SPSR)* of the mode from which the exception is taken.

The [MRS](#) and [MSR](#) instructions move the contents of the [CPSR](#), [APSR](#), or the [SPSR](#) of the current mode to or from a general-purpose register, see:

- [MRS](#).
- [MSR \(immediate\)](#).
- [MSR \(register\)](#).

When executed at EL0, [MRS](#) and [MSR](#) instructions can only access the [APSR](#).

The [PSTATE](#) Condition flags, [PSTATE](#).{N, Z, C, V} are set by the execution of data-processing instructions, and can control the execution of conditional instructions. However, software can set the Condition flags explicitly using the [MSR](#) instruction, and can read the current state of the Condition flags explicitly using the [MRS](#) instruction.

In addition, at EL1 or higher, software can use the [CPS](#) instruction to change the [PSTATE](#).M field and the [PSTATE](#).{A, I, F} interrupt mask bits, see [CPS](#), [CPSID](#), [CPSIE](#).

C2.5.2 Banked register access instructions

At EL1 or higher, the [MRS](#) (banked register) and [MSR](#) (banked register) instructions move the contents of a banked general-purpose register, the [SPSR](#), or the [ELR_hyp](#), to or from a general-purpose register. See:

- [MRS \(Banked register\)](#).
- [MSR \(Banked register\)](#).

C2.6 Load/store instructions

Table C2-12 summarizes the general-purpose register load/store instructions in the T32 and A32 instruction sets. Some of these instructions can also operate on the PC. See also:

- [Load/store multiple instructions](#).
- [Synchronization and semaphores](#), for more information about the Load-Exclusive and Store-Exclusive instructions.
- [Load-Acquire, Store-Release](#), for more information about the Load-Acquire/Store-Release and Load-Acquire Exclusive/Store-Release Exclusive instructions.
- [Advanced SIMD and floating-point load/store instructions](#).

Load/store instructions have several options for addressing memory. For more information, see [Addressing modes](#).

Table C2-12 Load/store instructions

Data type	Load	Store	Unprivileged		Exclusive		Load-Acquire	Store-Release	Exclusive Load-Acquire	Store-Release
			Load	Store	Load	Store				
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX	LDA	STL	LDAEX	STLEX
16-bit halfword	-	STRH	-	STRHT	-	STREXH	LDAH	STLH	LDAEXH	STLEXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-	-	-	-	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-	-	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB	LDAB	STLB	LDAEXB	STLEXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-	-	-	-	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-	-	-	-	-
Two 32-bit words	LDRD	STRD	-	-	-	-	-	-	-	-
64-bit doubleword	-	-	-	-	LDREXD	STREXD	-	-	LDAEXD	STLEXD

C2.6.1 Loads to the PC

The LDR instruction can load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

C2.6.2 Halfword and byte loads and stores

Halfword and byte stores store the least significant halfword or byte from the register, to 16 or 8 bits of memory respectively. There is no distinction between signed and unsigned stores.

Halfword and byte loads load 16 or 8 bits from memory into the least significant halfword or byte of a register. Unsigned loads zero-extend the loaded value to 32 bits, and signed loads sign-extend the value to 32 bits.

C2.6.3 Load unprivileged and Store unprivileged

When executing at EL0, a Load unprivileged or Store unprivileged instruction operates in the same way as the corresponding ordinary load or store instruction. For example, an LDRT instruction executes in exactly the same way as the equivalent LDR instruction. When executed at PL1, Load unprivileged and Store unprivileged instructions behave as

they would if they were executed at EL0. For example, an `LDRT` instruction executes in exactly the way that the equivalent `LDR` instruction would execute at EL0. In particular, the instructions make unprivileged memory accesses.

The Load unprivileged and Store unprivileged instructions are **CONSTRAINED UNPREDICTABLE** if executed at EL2.

For more information about execution privilege, see [Access permissions for instruction execution](#).

C2.6.4 Load-Exclusive and Store-Exclusive

Load-Exclusive and Store-Exclusive instructions provide shared memory synchronization. For more information, see [Synchronization and semaphores](#).

C2.6.5 Load-Acquire and Store-Release

Load-Acquire and Store-Release instructions provide memory barriers. Load-Acquire Exclusive and Store-Release Exclusive instructions provide memory barriers with shared memory synchronization. For more information, see [Load-Acquire, Store-Release](#).

C2.6.6 Addressing modes

The address for a load or store is formed from two parts: a value from a base register, and an offset.

The base register can be any one of the general-purpose registers R0-R12, SP, or LR.

For loads, the base register can be the PC. This provides PC-relative addressing for position-independent code. Instructions marked (literal) in their title in [About the T32 and A32 Instruction Descriptions](#) are PC-relative loads.

The offset takes one of three formats:

- | | |
|------------------------|---|
| Immediate | The offset is an unsigned number that can be added to or subtracted from the base register value. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets, and input/output registers. |
| Register | The offset is a value from a general-purpose register. The value can be added to, or subtracted from, the base register value. Register offsets are useful for accessing arrays or blocks of data. |
| Scaled register | The offset is a general-purpose register, shifted by an immediate value, then added to or subtracted from the base register. This means an array index can be scaled by the size of each array element. |

The offset and base register can be used in three different ways to form the memory address. The addressing modes are described as follows:

- | | |
|---------------------|--|
| Offset | The offset is added to or subtracted from the base register to form the memory address. |
| Pre-indexed | The offset is added to or subtracted from the base register to form the memory address. The base register is then updated with this new address, to permit automatic indexing through an array or memory block. |
| Post-indexed | The value of the base register alone is used as the memory address. The offset is then added to or subtracted from the base register. The result is stored back in the base register, to permit automatic indexing through an array or memory block. |

Note

Not every variant is available for every instruction, and the range of permitted immediate values and the options for scaled registers vary from instruction to instruction. See [About the T32 and A32 Instruction Descriptions](#) for full details for each instruction.

C2.7 Load/store multiple instructions

Load Multiple instructions load from memory a subset, or possibly all, of the general-purpose registers and the PC.

Store Multiple instructions store to memory a subset, or possibly all, of the general-purpose registers.

The memory locations are consecutive word-aligned words. The addresses used are obtained from a base register, and can be either above or below the value in the base register. The base register can optionally be updated by the total size of the data transferred.

[Table C2-13](#) summarizes the load/store multiple instructions in the T32 and A32 instruction sets.

Table C2-13 Load/store multiple instructions

Instruction	See
Load Multiple, Increment After or Full Descending	LDM, LDMIA, LDMFD
Load Multiple, Decrement After or Full Ascending ^a	LDMDA, LDMFA
Load Multiple, Decrement Before or Empty Ascending	LDMDB, LDMEA
Load Multiple, Increment Before or Empty Descending ^a	LDMIB, LDMED
Pop multiple registers off the stack ^b	POP
Push multiple registers onto the stack ^c	PUSH
Store Multiple, Increment After or Empty Ascending	STM, STMIA, STMEA
Store Multiple, Decrement After or Empty Descending ^a	STMDA, STMED
Store Multiple, Decrement Before or Full Descending	STMDB, STMFD
Store Multiple, Increment Before or Full Ascending ^a	STMIB, STMFA

a. Not available in the T32 instruction set.

b. This instruction is equivalent to an `LDM` instruction with the `SP` as base register, and base register updating.

c. This instruction is equivalent to an `STMDB` instruction with the `SP` as base register, and base register updating.

When executing at EL1, variants of the `LDM` and `STM` instructions load and store User mode registers. Another system level variant of the `LDM` instruction performs an exception return.

C2.7.1 Loads to the PC

The `LDM`, `LDMDA`, `LDMDB`, `LDMIB`, and `POP` instructions can load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

C2.8 Miscellaneous instructions

Table C2-14 summarizes the miscellaneous instructions in the T32 and A32 instruction sets.

Table C2-14 Miscellaneous instructions

Instruction	See
Clear-Exclusive	CLREX
Data Memory Barrier	DMB
Data Synchronization Barrier	DSB
Instruction Synchronization Barrier	ISB
If-Then	IT
No Operation	NOP
Preload Data	PLD, PLDW (immediate) PLD (literal) PLD, PLDW (register)
Preload Instruction	PLI (immediate, literal) PLI (register)
Set Endianness	SETEND ^a
Send Event	SEV
Send Event Local	SEVL
Wait For Event	WFE
Wait For Interrupt	WFI
Yield	YIELD ^b

a. Arm deprecates any use of the SETEND instruction.

b. See also [The Yield instruction](#).

———— Note ————

Previous versions of the architecture defined the DBG instruction, that could provide a hint to the debug system, in this group. From the introduction of Armv8, this instruction executes as a NOP. Arm deprecates any use of the DBG instruction.

C2.8.1 The Yield instruction

In a *Symmetric Multithreading* (SMT) design, a thread can use the YIELD instruction to give a hint to the PE that it is running on. The YIELD hint indicates that whatever the thread is currently doing is of low importance, and so could yield. For example, the thread might be sitting in a spin-lock. A similar use might be in modifying the arbitration priority of the snoop bus in a multiprocessor (MP) system. Defining such an instruction permits binary compatibility between SMT and SMP systems.

AArch32 state defines a YIELD instruction as a specific NOP (No Operation) hint instruction.

The YIELD instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use on migration to a multiprocessor or multithreading system. Operating systems can use YIELD in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

C2.9 Exception-generating and exception-handling instructions

The following instructions are intended specifically to cause a synchronous exception to occur:

- The `SVC` instruction generates a Supervisor Call exception. For more information, see [Supervisor Call \(SVC\) exception](#).
- The Breakpoint instruction `BKPT` provides software breakpoints. For more information, see [Breakpoint Instruction exceptions](#).
- In an implementation that includes EL2 the `HVC` instruction generates a Hypervisor Call exception. For more information, see [Hypervisor Call \(HVC\) exception](#).

[Debug state](#) summarizes the Debug state instructions.

For an exception taken to an EL1 mode:

- The system level variants of the `SUBS` and `LDM` instructions can perform a return from an exception.

Note

The variants of `SUBS` include `MOVS`. See the references to [Subtract \(exception return\)](#), [Move \(exception return\)](#), and [Load Multiple \(exception return\)](#) in [Table C2-15](#) for more information.

- The `SRS` instruction can be used near the start of the handler, to store return information. The `RFE` instruction can then perform a return from the exception using the stored return information.

In an implementation that includes EL2, the `ERET` instruction performs a return from an exception taken to Hyp mode.

For more information, see [Exception return to an Exception level using AArch32](#).

[Table C2-15](#) summarizes the instructions, in the T32 and A32 instruction sets, for generating or handling an exception. Except for `BKPT` and `SVC`, these are system level instructions.

Table C2-15 Exception-generating and exception-handling instructions

Instruction	See
Supervisor Call	<code>SVC</code>
Breakpoint	<code>BKPT</code>
Return From Exception	<code>RFE</code> , <code>RFEDA</code> , <code>RFEDB</code> , <code>RFEIA</code> , <code>RFEIB</code>
Subtract (exception return) ^a	<code>SUB</code> , <code>SUBS</code> (immediate) ^a
Move (exception return) ^a	<code>MOV</code> , <code>MOVS</code> (register) ^a
Hypervisor Call	<code>HVC</code>
Exception Return	<code>ERET</code>
Load Multiple (exception return)	<code>LDM</code> (exception return)
Store Return State	<code>SRS</code> , <code>SRSDA</code> , <code>SRSDB</code> , <code>SRSIA</code> , <code>SRSIB</code>

a. The A32 instruction set includes other instruction forms that can be used for an exception return, that have previously been described as variants of `SUBS` `PC`, `LR`. Arm deprecates any use of these instruction forms.

C2.9.1 Debug state

[Table C2-16](#) shows the Debug state instructions that are implemented in the T32 instruction set:

Table C2-16 T32 Debug state instructions

Mnemonic	Instruction	See	Note
DCPS _n	Debug switch to EL _n	DCPS1 DCPS2	-
ERET	Debug restore PE state (DRPS)	ERET	When executed in Debug state, the T1 encoding of ERET performs the DRPS operation

C2.10 System register access instructions

The System register encoding space is indexed using the parameters {coproc, opc1, CRn, CRm, opc2}, see [The AArch32 System register interface](#). This encoding space provides System registers and System instructions. The only permitted values of coproc are 0b1110 and 0b1111, and the following instructions give access to this encoding space:

- Instructions that transfer data between general-purpose registers and System registers. See:
 - [MCR](#).
 - [MCRR](#).
 - [MRC](#).
 - [MRRC](#).
- Instructions that load or store from memory to a System register. See:
 - [LDC \(immediate\)](#).
 - [LDC \(literal\)](#).
 - [STC](#).

Note

The System register encoding space with coproc == 0b101x is redefined to provide some of the Advanced SIMD and floating-point functionality. That is, to:

- Initiate a floating-point data-processing operation, see [Floating-point data-processing instructions](#).
 - Transfer data between general-purpose registers and the Advanced SIMD and floating-point registers, see [Advanced SIMD and floating-point register transfer instructions](#).
 - Load or store data to the Advanced SIMD and floating-point registers, see [Advanced SIMD and floating-point load/store instructions](#).
-

System register access instructions are part of the instruction stream executed by the PE, and therefore any System register access instruction that cannot be executed by the implementation causes an Undefined Instruction exception. In the A-profile and the R-profile architectures, the instruction encodings in the System register access instruction encoding space are unallocated, and generate Undefined Instruction exceptions, except for:

- The instructions summarized in this section that access the coproc == 0b111x encoding space.
- The instructions in the coproc == 0b101x encoding space that are redefined to provide Advanced SIMD and floating-point functionality, as summarized in the Note in this section.

C2.11 Advanced SIMD and floating-point load/store instructions

Table C2-17 summarizes the SIMD and floating-point register file load/store instructions in the Advanced SIMD and floating-point instruction sets.

Advanced SIMD also provides instructions for loading and storing multiple elements, or structures of elements, see [Element and structure load/store instructions](#).

Table C2-17 SIMD and floating-point register file load/store instructions

Instruction	See	Operation
Vector Load Multiple	VLDM , VLDMDB , VLDMIA	Load 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point. Load 1-32 consecutive 32-bit registers, floating-point only.
Vector Load Register	VLDR (immediate) VLDR (literal)	Load one 64-bit register, Advanced SIMD and floating-point. Load one 32-bit register, floating-point only.
Vector Store Multiple	VSTM , VSTMDB , VSTMIA	Store 1-16 consecutive 64-bit registers, Advanced SIMD and floating-point. Store 1-32 consecutive 32-bit registers, floating-point only.
Vector Store Register	VSTR	Store one 64-bit register, Advanced SIMD and floating-point. Store one 32-bit register, floating-point only.

C2.11.1 Element and structure load/store instructions

Table C2-18 shows the element and structure load/store instructions available in the Advanced SIMD instruction set. Loading and storing structures of more than one element automatically de-interleaves or interleaves the elements, see [Figure C2-1](#) for an example of de-interleaving. Interleaving is the inverse process.

Table C2-18 Element and structure load/store instructions

Instruction	See
Load single element	
Multiple elements	VLD1 (multiple single elements)
To one lane	VLD1 (single element to one lane)
To all lanes	VLD1 (single element to all lanes)
Load 2-element structure	
Multiple structures	VLD2 (multiple 2-element structures)
To one lane	VLD2 (single 2-element structure to one lane)
To all lanes	VLD2 (single 2-element structure to all lanes)
Load 3-element structure	
Multiple structures	VLD3 (multiple 3-element structures)
To one lane	VLD3 (single 3-element structure to one lane)
To all lanes	VLD3 (single 3-element structure to all lanes)

Instruction	See
Load 4-element structure	
Multiple structures	VLD4 (multiple 4-element structures)
To one lane	VLD4 (single 4-element structure to one lane)
To all lanes	VLD4 (single 4-element structure to all lanes)
Store single element	
Multiple elements	VST1 (multiple single elements)
From one lane	VST1 (single element from one lane)
Store 2-element structure	
Multiple structures	VST2 (multiple 2-element structures)
From one lane	VST2 (single 2-element structure from one lane)
Store 3-element structure	
Multiple structures	VST3 (multiple 3-element structures)
From one lane	VST3 (single 3-element structure from one lane)
Store 4-element structure	
Multiple structures	VST4 (multiple 4-element structures)
From one lane	VST4 (single 4-element structure from one lane)

Figure C2-1 shows the de-interleaving of a VLD3 . 16 (multiple 3-element structures) instruction:

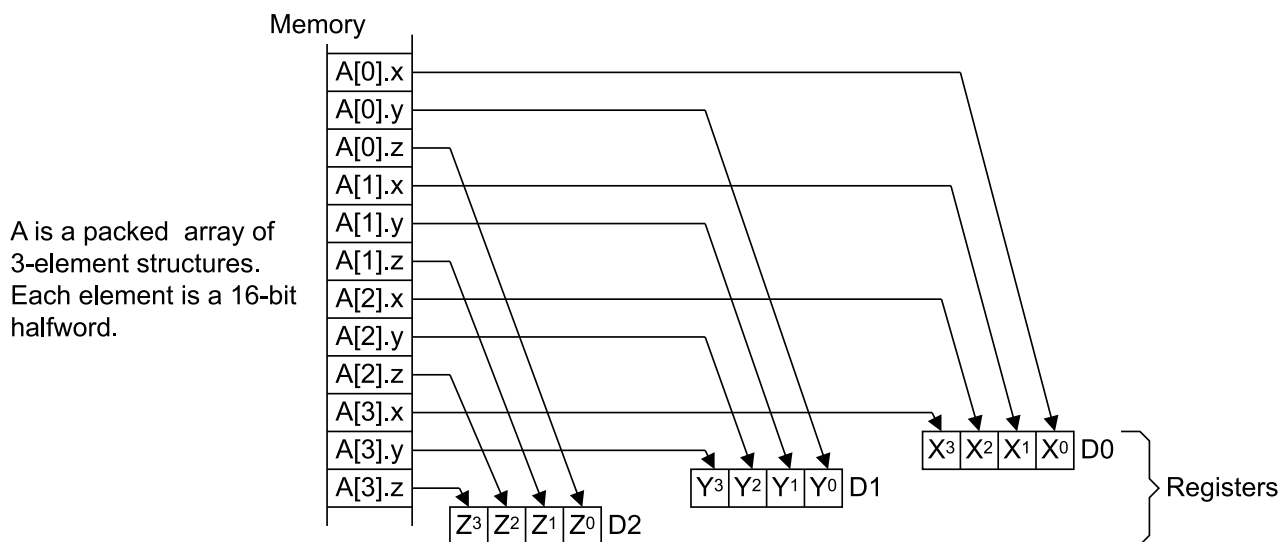


Figure C2-1 De-interleaving an array of 3-element structures

Figure C2-1 shows the VLD3 . 16 instruction operating to three 64-bit registers that comprise four 16-bit elements:

- Different instructions in this group would produce similar figures, but operate on different numbers of registers. For example, VLD4 and VST4 instructions operate on four registers.
- Different element sizes would produce similar figures but with 8-bit or 32-bit elements.
- These instructions operate only on doubleword (64-bit) registers.

C2.12 Advanced SIMD and floating-point register transfer instructions

[Table C2-19](#) summarizes the SIMD and floating-point register file transfer instructions in the Advanced SIMD and floating-point instruction sets. These instructions transfer data between the general-purpose registers and the registers in the SIMD and floating-point register file.

Advanced SIMD vectors, and single-precision and double-precision floating-point registers, are all views of the same register file. For details, see [The SIMD and floating-point register file](#).

Table C2-19 SIMD and floating-point register file transfer instructions

Instruction	See
Copy element from general-purpose register to every element of an Advanced SIMD vector	VDUP (general-purpose register)
Copy byte, halfword, or word from general-purpose register to a register in the SIMD and floating-point register file	VMOV (general-purpose register to scalar)
Copy byte, halfword, or word from a register in the SIMD and floating-point register file to a general-purpose register	VMOV (scalar to general-purpose register)
Copy from single-precision floating-point register to general-purpose register, or from general-purpose register to single-precision floating-point register	VMOV (between general-purpose register and single-precision)
Copy two words from general-purpose registers to consecutive single-precision floating-point registers, or from consecutive single-precision floating-point registers to general-purpose registers	VMOV (between two general-purpose registers and two single-precision registers)
Copy two words from general-purpose registers to a doubleword register in the SIMD and floating-point register file, or from a doubleword register in the SIMD and floating-point register file to general-purpose registers	VMOV (between two general-purpose registers and a doubleword floating-point register)
Copy from an Advanced SIMD and floating-point System Register to a general-purpose register	VMRS
Copy from a general-purpose register to an Advanced SIMD and floating-point System Register	VMSR

C2.13 Advanced SIMD data-processing instructions

Advanced SIMD data-processing instructions process registers containing vectors of elements of the same type packed together, enabling the same operation to be performed on multiple items in parallel.

Instructions operate on vectors held in 64-bit or 128-bit registers. [Figure C2-2](#) shows an operation on two 64-bit operand vectors, generating a 64-bit vector result.

Note

[Figure C2-2](#) and other similar figures show 64-bit vectors that consist of four 16-bit elements, and 128-bit vectors that consist of four 32-bit elements. Other element sizes produce similar figures, but with 1, 2, 8, or 16 operations performed in parallel instead of 4.

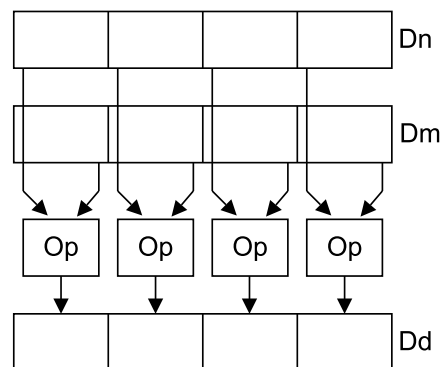


Figure C2-2 Advanced SIMD instruction operating on 64-bit registers

Many Advanced SIMD instructions have variants that produce vectors of elements double the size of the inputs. In this case, the number of elements in the result vector is the same as the number of elements in the operand vectors, but each element, and the whole vector, is double the size.

[Figure C2-3](#) shows an example of an Advanced SIMD instruction operating on 64-bit registers, and generating a 128-bit result.

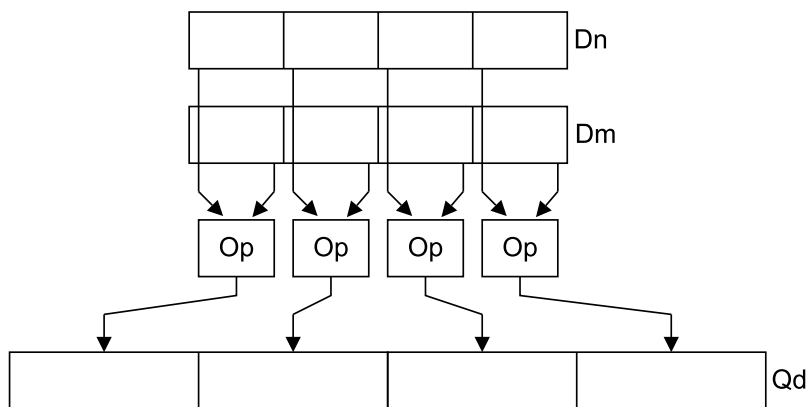


Figure C2-3 Advanced SIMD instruction producing wider result

There are also Advanced SIMD instructions that have variants that produce vectors containing elements half the size of the inputs. [Figure C2-4](#) shows an example of an Advanced SIMD instruction operating on one 128-bit register, and generating a 64-bit result.

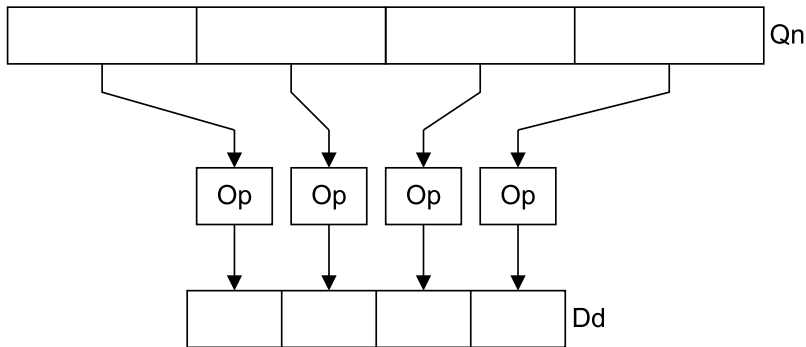


Figure C2-4 Advanced SIMD instruction producing narrower result

Some Advanced SIMD instructions do not conform to these standard patterns. Their operation patterns are described in the individual instruction descriptions.

Advanced SIMD instructions that perform floating-point arithmetic use the Arm standard floating-point arithmetic defined in [Floating-point support](#).

The following sections summarize the Advanced SIMD data-processing instructions:

- [Advanced SIMD parallel addition and subtraction](#).
- [Bitwise Advanced SIMD data-processing instructions](#).
- [Advanced SIMD comparison instructions](#).
- [Advanced SIMD shift instructions](#).
- [Advanced SIMD multiply instructions](#).
- [Miscellaneous Advanced SIMD data-processing instructions](#).
- [The Cryptographic Extension in AArch32 state](#).

C2.13.1 Advanced SIMD parallel addition and subtraction

Table C2-20 shows the Advanced SIMD parallel add and subtract instructions.

Table C2-20 Advanced SIMD parallel add and subtract instructions

Instruction	See
Vector Add	VADD (integer) VADD (floating-point)
Vector Add and Narrow, returning High Half	VADDHN
Vector Add Long	VADDL
Vector Add Wide	VADDW
Vector Halving Add	VHADD
Vector Halving Subtract	VHSUB
Vector Pairwise Add and Accumulate Long	VPADAL
Vector Pairwise Add	VPADD (integer) VPADD (floating-point)
Vector Pairwise Add Long	VPADDL
Vector Rounding Add and Narrow, returning High Half	VRADDHN

Instruction	See
Vector Rounding Halving Add	VRHADD
Vector Rounding Subtract and Narrow, returning High Half	VRSUBHN
Vector Saturating Add	VQADD
Vector Saturating Subtract	VQSUB
Vector Subtract	VSUB (integer) VSUB (floating-point)
Vector Subtract and Narrow, returning High Half	VSUBHN
Vector Subtract Long	VSUBL
Vector Subtract Wide	VSUBW

C2.13.2 Bitwise Advanced SIMD data-processing instructions

[Table C2-21](#) shows bitwise Advanced SIMD data-processing instructions. These operate on the doubleword (64-bit) or quadword (128-bit) registers in the SIMD and floating-point register file, and there is no division into vector elements.

Table C2-21 Bitwise Advanced SIMD data-processing instructions

Instruction	See
Vector Bitwise AND	VAND (register)
Vector Bitwise Bit Clear (AND complement)	VBIC (immediate) VBIC (register)
Vector Bitwise Exclusive OR	VEOR
Vector Bitwise Insert if False	VBIF
Vector Bitwise Insert if True	VBIT
Vector Bitwise Move	VMOV (immediate) VMOV (register)
Vector Bitwise NOT	VMVN (immediate) VMVN (register)
Vector Bitwise OR	VORR (immediate) VORR (register)
Vector Bitwise OR NOT	VORN (register)
Vector Bitwise Select	VBSL

C2.13.3 Advanced SIMD comparison instructions

[Table C2-22](#) shows Advanced SIMD comparison instructions.

Table C2-22 Advanced SIMD comparison instructions

Instruction	See
Vector Absolute Compare Greater Than or Equal	VACGE
Vector Absolute Compare Greater Than	VACGT
Vector Compare Equal	VCEQ (register)
Vector Compare Equal to Zero	VCEQ (immediate #0)
Vector Compare Greater Than or Equal	VCGE (register)
Vector Compare Greater Than or Equal to Zero	VCGE (immediate #0)
Vector Compare Greater Than	VCGT (register)
Vector Compare Greater Than Zero	VCGT (immediate #0)
Vector Compare Less Than or Equal to Zero	VCLE (immediate #0)
Vector Compare Less Than Zero	VCLT (immediate #0)
Vector Test Bits	VTST

C2.13.4 Advanced SIMD shift instructions

Table C2-23 lists the shift instructions in the Advanced SIMD instruction set.

Table C2-23 Advanced SIMD shift instructions

Instruction	See
Vector Saturating Rounding Shift Left	VQRSHL
Vector Saturating Rounding Shift Right and Narrow	VQRSHRN, VQRSHRUN
Vector Saturating Shift Left	VQSHL (register) VQSHL, VQSHLU (immediate)
Vector Saturating Shift Right and Narrow	VQSHRN, VQSHRUN
Vector Rounding Shift Left	VRSHL
Vector Rounding Shift Right	VRSHR
Vector Rounding Shift Right and Accumulate	VRSRA
Vector Rounding Shift Right and Narrow	VRSHRN
Vector Shift Left	VSHL (immediate) VSHL (register)
Vector Shift Left Long	VSHLL
Vector Shift Right	VSHR
Vector Shift Right and Narrow	VSHRN
Vector Shift Left and Insert	VSLI
Vector Shift Right and Accumulate	VSRA
Vector Shift Right and Insert	VSRI

C2.13.5 Advanced SIMD multiply instructions

Table C2-24 shows the Advanced SIMD multiply instructions.

Table C2-24 Advanced SIMD multiply instructions

Instruction	See
Vector Multiply Accumulate	VMLA (integer) VMLA (floating-point) VMLA (by scalar)
Vector Multiply Accumulate Long	VMLAL (integer) VMLAL (by scalar)
Vector Multiply Subtract	VMLS (integer) VMLS (floating-point) VMLS (by scalar)
Vector Multiply Subtract Long	VMLSL (integer) VMLSL (by scalar)
Vector Multiply	VMUL (integer and polynomial) VMUL (floating-point) VMUL (by scalar)
Vector Multiply Long	VMULL (integer and polynomial) VMULL (by scalar)
Vector Fused Multiply Accumulate	VFMA
Vector Fused Multiply Subtract	VFMS
Vector Saturating Doubling Multiply Accumulate Long	VQDMLAL
Vector Saturating Doubling Multiply Subtract Long	VQDMLSL
Vector Saturating Doubling Multiply Returning High Half	VQDMULH
Vector Saturating Doubling Multiply Long	VQDMULL
Vector Saturating Rounding Doubling Multiply Returning High Half	VQRDMULH

Advanced SIMD multiply instructions can operate on vectors of:

- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit polynomials over $\{0, 1\}$. VMUL and VMULL are the only instructions that operate on polynomials. VMULL produces a 16-bit polynomial over $\{0, 1\}$.
- Single-precision (32-bit) floating-point numbers.

They can also act on one vector and one scalar.

Long instructions have doubleword (64-bit) operands, and produce quadword (128-bit) results. Other Advanced SIMD multiply instructions can have either doubleword or quadword operands, and produce results of the same size.

Floating-point multiply instructions can operate on:

- Single-precision (32-bit) floating-point numbers.
- Double-precision (64-bit) floating-point numbers.

C2.13.6 Miscellaneous Advanced SIMD data-processing instructions

Table C2-25 shows miscellaneous Advanced SIMD data-processing instructions.

Table C2-25 Miscellaneous Advanced SIMD data-processing instructions

Instruction	See
Vector Absolute Difference and Accumulate	VABA
Vector Absolute Difference and Accumulate Long	VABAL
Vector Absolute Difference	VABD (integer) VABD (floating-point)
Vector Absolute Difference Long	VABDL (integer)
Vector Absolute	VABS
Vector Convert between floating-point and fixed-point	VCVT (between floating-point and fixed-point, Advanced SIMD)
Vector Convert between floating-point and integer	VCVT (between floating-point and integer, Advanced SIMD)
Vector Count Leading Sign Bits	VCLS
Vector Count Leading Zeros	VCLZ
Vector Count Set Bits	VCNT
Vector Duplicate scalar	VDUP (scalar)
Vector Extract	VEXT (byte elements)
Vector Move and Narrow	VMOVN
Vector Move Long	VMOVL
Vector Maximum	VMAX (integer) VMAX (floating-point)
Vector Minimum	VMIN (integer) VMIN (floating-point)
Vector Negate	VNEG
Vector Pairwise Maximum	VPMAX (integer) VPMAX (floating-point)
Vector Pairwise Minimum	VPMIN (integer) VPMIN (floating-point)
Vector Reciprocal Estimate	VRECPE
Vector Reciprocal Step	VRECPS
Vector Reciprocal Square Root Estimate	VRSQRTE
Vector Reciprocal Square Root Step	VRSQRTS
Vector Reverse in halfwords	VREV16
Vector Reverse in words	VREV32
Vector Reverse in doublewords	VREV64
Vector Saturating Absolute	VQABS
Vector Saturating Move and Narrow	VQMOVN, VQMOVUN
Vector Saturating Negate	VQNEG
Vector Swap	VSWP
Vector Table Lookup	VTBL, VTBX
Vector Transpose	VTRN
Vector Unzip	VUZP
Vector Zip	VZIP

Instruction	See
-------------	-----

C2.13.7 The Cryptographic Extension in AArch32 state

The instructions provided by the optional Cryptographic Extension use the Advanced SIMD and floating-point register file. For more information about the functions they provide see:

- *Advanced Encryption Standard (AES).*
- *The Galois/Counter Mode of Operation.*
- *Secure Hash Algorithm (SHA).*

Table C2-26 shows the AArch32 Cryptographic Extension instructions.

Table C2-26 AArch32 Cryptographic Extension instructions

Mnemonic	Instruction	See
AESD	AES single round decryption	AESD
AESE	AES single round encryption	AESE
AESIMC	AES inverse mix columns	AESIMC
AESMC	AES mix columns	AESMC
VMULL	Polynomial multiply long	VMULL (integer and polynomial) ^a
SHA1C	SHA1 hash update (choose)	SHA1C
SHA1H	SHA1 fixed rotate	SHA1H
SHA1M	SHA1 hash update (majority)	SHA1M
SHA1P	SHA1 hash update (parity)	SHA1P
SHA1SU0	SHA1 schedule update 0	SHA1SU0
SHA1SU1	SHA1 schedule update 1	SHA1SU1
SHA256H	SHA256 hash update (part 1)	SHA256H
SHA256H2	SHA256 hash update (part 2)	SHA256H2
SHA256SU0	SHA256 schedule update 0	SHA256SU0
SHA256SU1	SHA256 schedule update 1	SHA256SU1

a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

C2.14 Floating-point data-processing instructions

[Table C2-27](#) summarizes the data-processing instructions in the floating-point instruction set. In this table, *floating-point register* means a register in the SIMD and floating-point register file.

For details of the floating-point arithmetic used by floating-point instructions, see [Floating-point support](#).

Table C2-27 Floating-point data-processing instructions

Instruction	See
Convert between double-precision and single-precision	VCVT (between double-precision and single-precision)
Convert between floating-point and fixed-point	VCVT (between floating-point and fixed-point, floating-point)
Convert between half-precision and single-precision, writing to bottom half of single-precision register	VCVTB
Convert between half-precision and single-precision, writing to top half of single-precision register	VCVTT
Convert from floating-point to integer	VCVT (floating-point to integer, floating-point)
Convert from floating-point to integer using FPSCR rounding mode	VCVTR
Convert from integer to floating-point	VCVT (integer to floating-point, floating-point)
Copy from one floating-point register to another	VMOV (register)
Divide	VDIV
Move immediate value to a floating-point register	VMOV (immediate)
Square Root	VSQRT
Vector Absolute value	VABS
Vector Add	VADD (floating-point)
Vector Compare with exceptions disabled	VCMPE
Vector Compare with exceptions enabled	VCMP
Vector Fused Multiply Accumulate	VFMA
Vector Fused Multiply Subtract	VFMS
Vector Fused Negate Multiply Accumulate	VFNMA
Vector Fused Negate Multiply Subtract	VFNMS
Vector Multiply	VMUL (floating-point)
Vector Multiply Accumulate	VMLA (floating-point)
Vector Multiply Subtract	VMLS (floating-point)
Vector Negate Multiply	VNMUL
Vector Negate Multiply Accumulate	VNMLA
Vector Negate Multiply Subtract	VNMLS
Vector Negate, by inverting the sign bit	VNEG
Vector Subtract	VSUB (floating-point)

Chapter C3

T32 Instruction Set Encoding

This chapter describes the encoding of the T32 instruction set. It contains the following sections:

- [T32 instruction set encoding](#)
- [About the T32 Advanced SIMD and floating-point instructions and their encoding](#)

In this chapter:

- In the decode tables, an entry of $\times$ for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.

C3.1 T32 instruction set encoding

The T32 instruction stream is a sequence of halfword-aligned halfwords. Each T32 instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If the value of bits[15:11] of the halfword being decoded is one of the following, the halfword is the first halfword of a 32-bit instruction:

- 0b11101.
- 0b11110.
- 0b11111.

Otherwise, the halfword is a 16-bit instruction.

The T32 instruction encoding is:

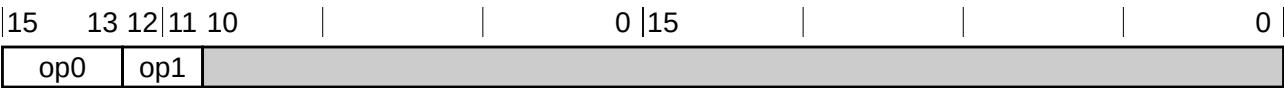


Table C3-1 Main encoding table for the T32 instruction set

Decode fields		Decode group or instruction page
op0	op1	
!= 111	xx	16-bit
111	00	B - T2 variant
111	!= 00	32-bit

C3.1.1 16-bit

This section describes the encoding of the 16-bit group. The encodings in this section are decoded from [T32 instruction set encoding](#).



This decode also imposes the constraint:

- op0<5:3> != 111.

Table C3-2 Encoding table for the 16-bit group

Decode fields	Decode group or instruction page
op0	
00xxxx	Shift (immediate), add, subtract, move, and compare
010000	Data-processing (two low registers)
010001	Special data instructions and branch and exchange
01001x	LDR (literal) - T1 variant
0101xx	Load/store (register offset)
011xxx	Load/store word/byte (immediate offset)
1000xx	Load/store halfword (immediate offset)
1001xx	Load/store (SP-relative)
1010xx	Add PC/SP (immediate)
1011xx	Miscellaneous 16-bit instructions
1100xx	Load/store multiple
1101xx	Conditional branch, and Supervisor Call

C3.1.1.1 Data-processing (two low registers)

This section describes the encoding of the Data-processing (two low registers) instruction class. The encodings in this section are decoded from 16-bit.

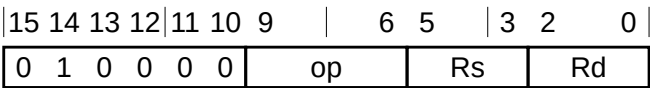


Table C3-3

Decode fields	Instruction page
op	
0000	AND, ANDS (register)
0001	EOR, EORS (register)
0010	MOV, MOVS (register-shifted register) - Logical shift left variant
0011	MOV, MOVS (register-shifted register) - Logical shift right variant
0100	MOV, MOVS (register-shifted register) - Arithmetic shift right variant
0101	ADC, ADCS (register)
0110	SBC, SBCS (register)
0111	MOV, MOVS (register-shifted register) - Rotate right variant
1000	TST (register)
1001	RSB, RSBS (immediate)

Decode fields	Instruction page
op	
1010	CMP (register)
1011	CMN (register)
1100	ORR, ORRS (register)
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

C3.1.1.2 Load/store (register offset)

This section describes the encoding of the Load/store (register offset) instruction class. The encodings in this section are decoded from 16-bit.

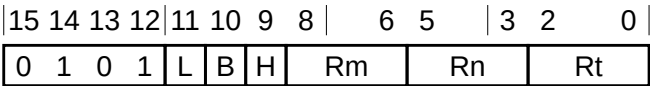


Table C3-4

Decode fields			Instruction page
L	B	H	
0	0	0	STR (register)
0	0	1	STRH (register)
0	1	0	STRB (register)
0	1	1	LDRSB (register)
1	0	0	LDR (register)
1	0	1	LDRH (register)
1	1	0	LDRB (register)
1	1	1	LDRSH (register)

C3.1.1.3 Load/store word/byte (immediate offset)

This section describes the encoding of the Load/store word/byte (immediate offset) instruction class. The encodings in this section are decoded from 16-bit.

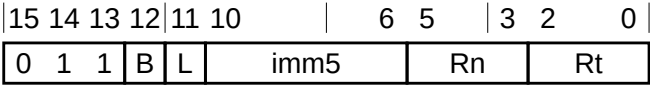


Table C3-5

Decode fields		Instruction page
B	L	
0	0	STR (immediate)
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

C3.1.1.4 Load/store halfword (immediate offset)

This section describes the encoding of the Load/store halfword (immediate offset) instruction class. The encodings in this section are decoded from 16-bit.

15	14	13	12	11	10		6	5		3	2	0
1	0	0	0	L		imm5			Rn			Rt

Table C3-6

Decode fields	Instruction page
L	
0	STRH (immediate)
1	LDRH (immediate)

C3.1.1.5 Load/store (SP-relative)

This section describes the encoding of the Load/store (SP-relative) instruction class. The encodings in this section are decoded from 16-bit.

15	14	13	12	11	10	8	7				0
1	0	0	1	L		Rt				imm8	

Table C3-7

Decode fields	Instruction page
L	
0	STR (immediate)
1	LDR (immediate)

C3.1.1.6 Add PC/SP (immediate)

This section describes the encoding of the Add PC/SP (immediate) instruction class. The encodings in this section are decoded from 16-bit.

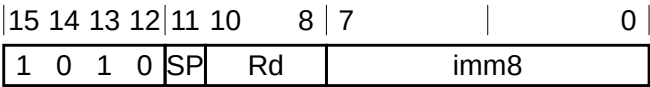


Table C3-8

Decode fields	Instruction page
SP	
0	ADR
1	ADD, ADDS (SP plus immediate)

C3.1.1.7 Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from 16-bit.

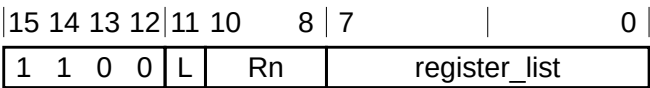


Table C3-9

Decode fields	Instruction page
L	
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

C3.1.2 Shift (immediate), add, subtract, move, and compare

This section describes the encoding of the Shift (immediate), add, subtract, move, and compare group. The encodings in this section are decoded from 16-bit.

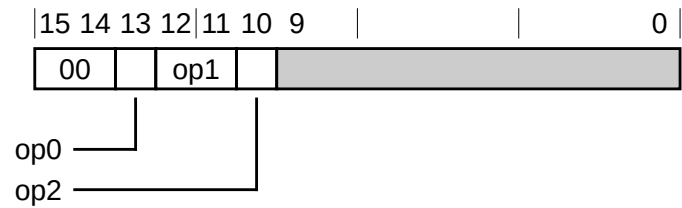


Table C3-10 Encoding table for the Shift (immediate), add, subtract, move, and compare group

Decode fields			Decode group or instruction page
op0	op1	op2	
0	11	0	Add, subtract (three low registers)
0	11	1	Add, subtract (two low registers and immediate)
0	!= 11	x	MOV, MOVS (register) - T2 variant
1	xx	x	Add, subtract, compare, move (one low register and immediate)

C3.1.2.1 Add, subtract (three low registers)

This section describes the encoding of the Add, subtract (three low registers) instruction class. The encodings in this section are decoded from Shift (immediate), add, subtract, move, and compare.

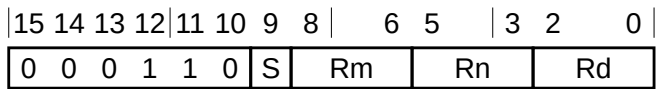


Table C3-11

Decode fields	Instruction page
S	
0	ADD, ADDS (register)
1	SUB, SUBS (register)

C3.1.2.2 Add, subtract (two low registers and immediate)

This section describes the encoding of the Add, subtract (two low registers and immediate) instruction class. The encodings in this section are decoded from Shift (immediate), add, subtract, move, and compare.

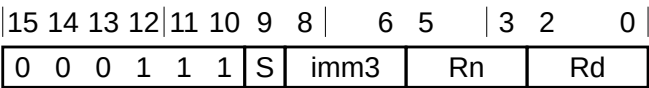


Table C3-12

Decode fields	Instruction page
S	
0	ADD, ADDS (immediate)
1	SUB, SUBS (immediate)

C3.1.2.3 Add, subtract, compare, move (one low register and immediate)

This section describes the encoding of the Add, subtract, compare, move (one low register and immediate) instruction class. The encodings in this section are decoded from [Shift \(immediate\)](#), [add](#), [subtract](#), [move](#), and [compare](#).

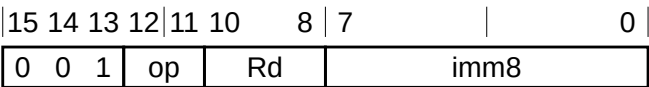


Table C3-13

Decode fields	Instruction page
op	
00	MOV, MOVS (immediate)
01	CMP (immediate)
10	ADD, ADDS (immediate)
11	SUB, SUBS (immediate)

C3.1.3 Special data instructions and branch and exchange

This section describes the encoding of the Special data instructions and branch and exchange group. The encodings in this section are decoded from [16-bit](#).

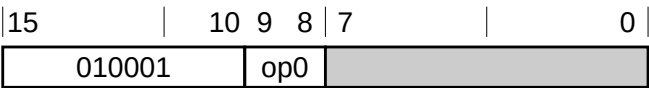


Table C3-14 Encoding table for the Special data instructions and branch and exchange group

Decode fields	Decode group or instruction page
op0	
11	Branch and exchange
!= 11	Add, subtract, compare, move (two high registers)

C3.1.3.1 Branch and exchange

This section describes the encoding of the Branch and exchange instruction class. The encodings in this section are decoded from Special data instructions and branch and exchange.

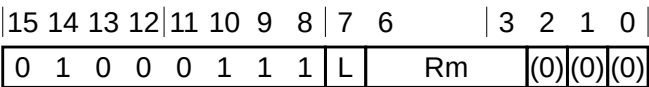


Table C3-15

Decode fields	Instruction page
L	
0	BX
1	BLX (register)

C3.1.3.2 Add, subtract, compare, move (two high registers)

This section describes the encoding of the Add, subtract, compare, move (two high registers) instruction class. The encodings in this section are decoded from Special data instructions and branch and exchange.

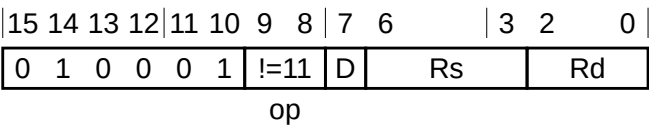


Table C3-16

Decode fields			Instruction page
op	D:Rd	Rs	
00	!= 1101	!= 1101	ADD, ADDS (register)
00	xxxx	1101	ADD, ADDS (SP plus register) - T1
00	1101	!= 1101	ADD, ADDS (SP plus register) - T2
01	xxxx	xxxx	CMP (register)
10	xxxx	xxxx	MOV, MOVS (register)

C3.1.4 Miscellaneous 16-bit instructions

This section describes the encoding of the Miscellaneous 16-bit instructions group. The encodings in this section are decoded from [16-bit](#).

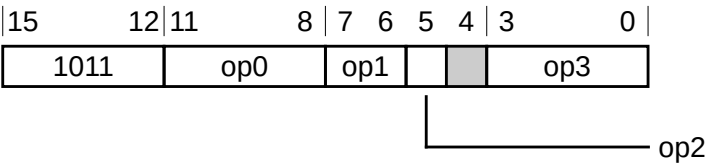


Table C3-17 Encoding table for the Miscellaneous 16-bit instructions group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0000	xx	x	xxxx	Adjust SP (immediate)
0010	xx	x	xxxx	Extend
0110	! =01	x	xxxx	Unallocated.
0110	01	x	xxxx	Change Processor State
1010	10	x	xxxx	HLT
1010	!= 10	x	xxxx	Reverse bytes
1110	xx	x	xxxx	BKPT
1111	xx	x	0000	Hints
1111	xx	x	!= 0000	IT
x0x1	xx	x	xxxx	CBNZ, CBZ
x10x	xx	x	xxxx	Push and Pop

C3.1.4.1 Adjust SP (immediate)

This section describes the encoding of the Adjust SP (immediate) instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

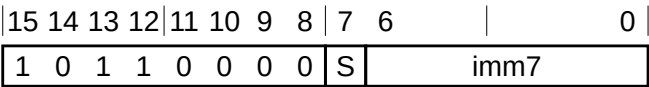


Table C3-18

Decode fields	Instruction page
S	
0	ADD, ADDS (SP plus immediate)
1	SUB, SUBS (SP minus immediate)

C3.1.4.2 Extend

This section describes the encoding of the Extend instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

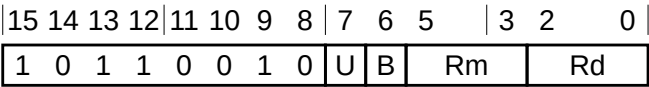


Table C3-19

Decode fields		Instruction page
U	B	
0	0	SXTH
0	1	SXTB
1	0	UXTH
1	1	UXTB

C3.1.4.3 Change Processor State

This section describes the encoding of the Change Processor State instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

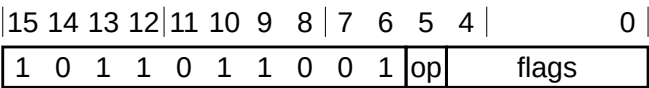


Table C3-20

Decode fields		Instruction page
op	flags	
0	xxxxx	SETEND
1	0xxxx	CPS, CPSID, CPSIE - Interrupt enable variant
1	1xxxx	CPS, CPSID, CPSIE - Interrupt disable variant

C3.1.4.4 Reverse bytes

This section describes the encoding of the Reverse bytes instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions.

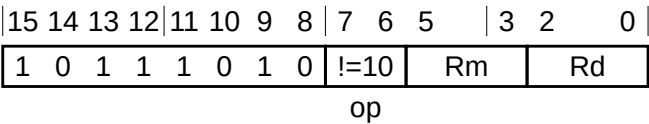


Table C3-21

Decode fields		Instruction page
op		
00		REV
01		REV16
11		REVSH

C3.1.4.5 Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from Miscellaneous 16-bit instructions.

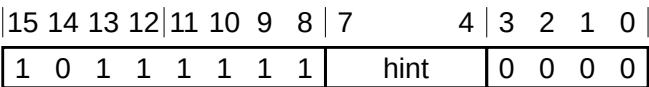


Table C3-22

Decode fields	Instruction page
hint	
0000	NOP
0001	YIELD
0010	WFE
0011	WFI
0100	SEV
0101	SEVL
011x	Reserved hint, behaves as NOP.
1xxx	Reserved hint, behaves as NOP.

C3.1.4.6 Push and Pop

This section describes the encoding of the Push and Pop instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

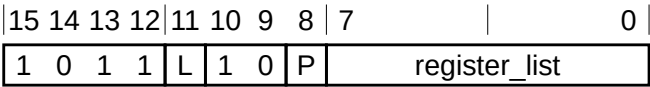


Table C3-23

Decode fields	Instruction page
L	
0	PUSH
1	POP

C3.1.5 Conditional branch, and Supervisor Call

This section describes the encoding of the Conditional branch, and Supervisor Call group. The encodings in this section are decoded from [16-bit](#).

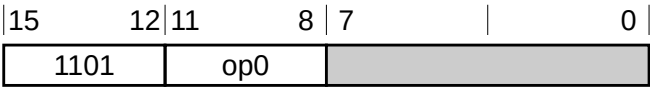


Table C3-24 Encoding table for the Conditional branch, and Supervisor Call group

Decode fields	Decode group or instruction page
op0	
111x	Exception generation
!= 111x	B - T1 variant

C3.1.5.1 Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from Conditional branch, and Supervisor Call.

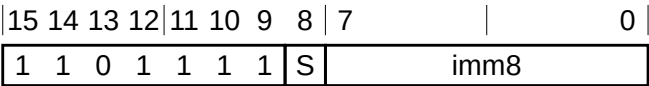
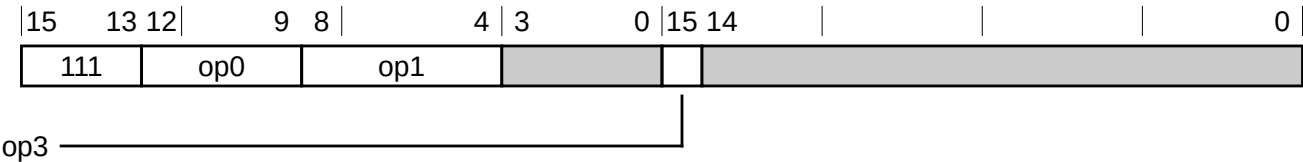


Table C3-25

Decode fields	Instruction page
S	
0	UDF
1	SVC

C3.1.6 32-bit

This section describes the encoding of the 32-bit group. The encodings in this section are decoded from T32 instruction set encoding.



This decode also imposes the constraint:

- op0<3:2> != 00.

Table C3-26 Encoding table for the 32-bit group

Decode fields			Decode group or instruction page
op0	op1	op3	
x11x	xxxxx	x	System register access, Advanced SIMD, and floating-point
0100	xx0xx	x	Load/store multiple
0100	xx1xx	x	Load/store dual, load/store exclusive, load-acquire/store-release, and table branch
0101	xxxxx	x	Data-processing (shifted register)
10xx	xxxxx	1	Branches and miscellaneous control
10x0	xxxxx	0	Data-processing (modified immediate)
10x1	xxxx0	0	Data-processing (plain binary immediate)
10x1	xxxx1	0	Unallocated.
1100	1xxx0	x	Advanced SIMD element or structure load/store
1100	!= 1xxx0	x	Load/store single
1101	0xxxx	x	Data-processing (register)
1101	10xxx	x	Multiply, multiply accumulate, and absolute difference
1101	11xxx	x	Long multiply and divide

C3.1.6.1 Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from 32-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	14	13					0
1	1	1	0	1	0	0	opc	0	W	L		Rn	P	M							register_list

Table C3-27

Decode fields		Instruction page
opc	L	
00	0	SRS, SRSDA, SRSDb, SRSIA, SRSIB - T1
00	1	RFE, RFEDA, RFEDb, RFEIA, RFEIB - T1
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	0	SRS, SRSDA, SRSDb, SRSIA, SRSIB - T2

Decode fields		Instruction page
opc	L	
11	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - T2

C3.1.6.2 Data-processing (shifted register)

This section describes the encoding of the Data-processing (shifted register) instruction class. The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	5	4	3	0	15	14	12	11	8	7	6	5	4	3	0
1	1	1	0	1	0	1		op1	S	Rn	(0)	imm3	Rd	imm2	stype	Rm						

Table C3-28

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:stype	Rd	
0000	0	xxxx	!= 0000011	xxxx	AND, ANDS (register) - AND, shift or rotate by value variant
0000	0	xxxx	0000011	xxxx	AND, ANDS (register) - AND, rotate right with extend variant
0000	1	xxxx	!= 0000011	!= 1111	AND, ANDS (register) - ANDS, shift or rotate by value variant
0000	1	xxxx	!= 0000011	1111	TST (register) - Shift or rotate by value variant
0000	1	xxxx	0000011	!= 1111	AND, ANDS (register) - ANDS, rotate right with extend variant
0000	1	xxxx	0000011	1111	TST (register) - Rotate right with extend variant
0001	x	xxxx	!= 0000011	xxxx	BIC, BICS (register) - BICS, shift or rotate by value variant
0001	x	xxxx	0000011	xxxx	BIC, BICS (register) - BICS, rotate right with extend variant
0010	0	!= 1111	!= 0000011	xxxx	ORR, ORRS (register) - ORR, shift or rotate by value variant
0010	0	!= 1111	0000011	xxxx	ORR, ORRS (register) - ORR, rotate right with extend variant
0010	0	1111	!= 0000011	xxxx	MOV, MOVS (register) - MOV, shift or rotate by value variant
0010	0	1111	0000011	xxxx	MOV, MOVS (register) - MOV, rotate right with extend variant
0010	1	!= 1111	!= 0000011	xxxx	ORR, ORRS (register) - ORRS, shift or rotate by value variant
0010	1	!= 1111	0000011	xxxx	ORR, ORRS (register) - ORRS, rotate right with extend variant
0010	1	1111	!= 0000011	xxxx	MOV, MOVS (register) - MOVS, shift or rotate by value variant
0010	1	1111	0000011	xxxx	MOV, MOVS (register) - MOVS, rotate right with extend variant
0011	0	!= 1111	!= 0000011	xxxx	ORN, ORNS (register) - ORN, shift or rotate by value variant
0011	0	!= 1111	0000011	xxxx	ORN, ORNS (register) - ORN, rotate right with extend variant
0011	0	1111	!= 0000011	xxxx	MVN, MVNS (register) - MVN, shift or rotate by value variant
0011	0	1111	0000011	xxxx	MVN, MVNS (register) - MVN, rotate right with extend variant
0011	1	!= 1111	!= 0000011	xxxx	ORN, ORNS (register) - ORNS, shift or rotate by value variant
0011	1	!= 1111	0000011	xxxx	ORN, ORNS (register) - ORNS, rotate right with extend variant

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:stype	Rd	
0011	1	1111	!= 0000011	xxxx	MVN, MVNS (register) - MVNS, shift or rotate by value variant
0011	1	1111	0000011	xxxx	MVN, MVNS (register) - MVNS, rotate right with extend variant
0100	0	xxxx	!= 0000011	xxxx	EOR, EORS (register) - EOR, shift or rotate by value variant
0100	0	xxxx	0000011	xxxx	EOR, EORS (register) - EOR, rotate right with extend variant
0100	1	xxxx	!= 0000011	!= 1111	EOR, EORS (register) - EORS, shift or rotate by value variant
0100	1	xxxx	!= 0000011	1111	TEQ (register) - Shift or rotate by value variant
0100	1	xxxx	0000011	!= 1111	EOR, EORS (register) - EORS, rotate right with extend variant
0100	1	xxxx	0000011	1111	TEQ (register) - Rotate right with extend variant
0101	x	xxxx	xxxxxxx	xxxx	Unallocated.
0110	0	xxxx	xxxxxx00	xxxx	PKHBT, PKHTB - PKHBT variant
0110	0	xxxx	xxxxxx01	xxxx	Unallocated.
0110	0	xxxx	xxxxxx10	xxxx	PKHBT, PKHTB - PKHTB variant
0110	0	xxxx	xxxxxx11	xxxx	Unallocated.
0111	x	xxxx	xxxxxxx	xxxx	Unallocated.
1000	0	!= 1101	!= 0000011	xxxx	ADD, ADDS (register) - ADD, shift or rotate by value variant
1000	0	!= 1101	0000011	xxxx	ADD, ADDS (register) - ADD, rotate right with extend variant
1000	0	1101	!= 0000011	xxxx	ADD, ADDS (SP plus register) - ADD, shift or rotate by value variant
1000	0	1101	0000011	xxxx	ADD, ADDS (SP plus register) - ADD, rotate right with extend variant
1000	1	xxxx	!= 0000011	1111	CMN (register) - Shift or rotate by value variant
1000	1	!= 1101	!= 0000011	!= 1111	ADD, ADDS (register) - ADDS, shift or rotate by value variant
1000	1	!= 1101	0000011	!= 1111	ADD, ADDS (register) - ADDS, rotate right with extend variant
1000	1	xxxx	0000011	1111	CMN (register) - Rotate right with extend variant
1000	1	1101	!= 0000011	!= 1111	ADD, ADDS (SP plus register) - ADDS, shift or rotate by value variant
1000	1	1101	0000011	!= 1111	ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant
1001	x	xxxx	xxxxxxx	xxxx	Unallocated.
1010	x	xxxx	!= 0000011	xxxx	ADC, ADCS (register) - ADCS, shift or rotate by value variant
1010	x	xxxx	0000011	xxxx	ADC, ADCS (register) - ADCS, rotate right with extend variant
1011	x	xxxx	!= 0000011	xxxx	SBC, SBCS (register) - SBCS, shift or rotate by value variant
1011	x	xxxx	0000011	xxxx	SBC, SBCS (register) - SBCS, rotate right with extend variant
1100	x	xxxx	xxxxxxx	xxxx	Unallocated.
1101	0	!= 1101	!= 0000011	xxxx	SUB, SUBS (register) - SUB, shift or rotate by value variant
1101	0	!= 1101	0000011	xxxx	SUB, SUBS (register) - SUB, rotate right with extend variant
1101	0	1101	!= 0000011	xxxx	SUB, SUBS (SP minus register) - SUB, shift or rotate by value variant
1101	0	1101	0000011	xxxx	SUB, SUBS (SP minus register) - SUB, rotate right with extend variant
1101	1	xxxx	!= 0000011	1111	CMP (register) - Shift or rotate by value variant

Decode fields				Instruction page	
op1	S	Rn	imm3:imm2:stype	Rd	
1101	1	!= 1101	!= 0000011	!= 1111	SUB, SUBS (register) - SUBS, shift or rotate by value variant
1101	1	!= 1101	0000011	!= 1111	SUB, SUBS (register) - SUBS, rotate right with extend variant
1101	1	xxxx	0000011	1111	CMP (register) - Rotate right with extend variant
1101	1	1101	!= 0000011	!= 1111	SUB, SUBS (SP minus register) - SUBS, shift or rotate by value variant
1101	1	1101	0000011	!= 1111	SUB, SUBS (SP minus register) - SUBS, rotate right with extend variant
1110	x	xxxx	!= 0000011	xxxx	RSB, RSBS (register) - RSBS, shift or rotate by value variant
1110	x	xxxx	0000011	xxxx	RSB, RSBS (register) - RSBS, rotate right with extend variant
1111	x	xxxx	xxxxxxxx	xxxx	Unallocated.

C3.1.6.3 Data-processing (modified immediate)

This section describes the encoding of the Data-processing (modified immediate) instruction class. The encodings in this section are decoded from [32-bit](#).

15	14	13	12	11	10	9	8	5	4	3	0	15	14	12	11	8	7	0
1	1	1	1	0	i	0		op1	S	Rn	0	imm3		Rd				imm8

Table C3-29

Decode fields				Instruction page	
op1	S	Rn	Rd		
0000	0	xxxx	xxxx	AND, ANDS (immediate) - AND variant	
0000	1	xxxx	!= 1111	AND, ANDS (immediate) - ANDS variant	
0000	1	xxxx	1111	TST (immediate)	
0001	x	xxxx	xxxx	BIC, BICS (immediate)	
0010	0	!= 1111	xxxx	ORR, ORRS (immediate) - ORR variant	
0010	0	1111	xxxx	MOV, MOVS (immediate) - MOV variant	
0010	1	!= 1111	xxxx	ORR, ORRS (immediate) - ORRS variant	
0010	1	1111	xxxx	MOV, MOVS (immediate) - MOVS variant	
0011	0	!= 1111	xxxx	ORN, ORNS (immediate) - Not flag setting variant	
0011	0	1111	xxxx	MVN, MVNS (immediate) - MVN variant	
0011	1	!= 1111	xxxx	ORN, ORNS (immediate) - Flag setting variant	
0011	1	1111	xxxx	MVN, MVNS (immediate) - MVNS variant	
0100	0	xxxx	xxxx	EOR, EORS (immediate) - EOR variant	
0100	1	xxxx	!= 1111	EOR, EORS (immediate) - EORS variant	

Decode fields				Instruction page
op1	S	Rn	Rd	
0100	1	xxxx	1111	TEQ (immediate)
0101	x	xxxx	xxxx	Unallocated.
011x	x	xxxx	xxxx	Unallocated.
1000	0	!= 1101	xxxx	ADD, ADDS (immediate) - ADD variant
1000	0	1101	xxxx	ADD, ADDS (SP plus immediate) - ADD variant
1000	1	!= 1101	!= 1111	ADD, ADDS (immediate) - ADDS variant
1000	1	1101	!= 1111	ADD, ADDS (SP plus immediate) - ADDS variant
1000	1	xxxx	1111	CMN (immediate)
1001	x	xxxx	xxxx	Unallocated.
1010	x	xxxx	xxxx	ADC, ADCS (immediate)
1011	x	xxxx	xxxx	SBC, SBCS (immediate)
1100	x	xxxx	xxxx	Unallocated.
1101	0	!= 1101	xxxx	SUB, SUBS (immediate) - SUB variant
1101	0	1101	xxxx	SUB, SUBS (SP minus immediate) - SUB variant
1101	1	!= 1101	!= 1111	SUB, SUBS (immediate) - SUBS variant
1101	1	1101	!= 1111	SUB, SUBS (SP minus immediate) - SUBS variant
1101	1	xxxx	1111	CMP (immediate)
1110	x	xxxx	xxxx	RSB, RSBS (immediate)
1111	x	xxxx	xxxx	Unallocated.

C3.1.6.4 Long multiply and divide

This section describes the encoding of the Long multiply and divide instruction class. The encodings in this section are decoded from 32-bit.

15	14	13	12	11	10	9	8	7	6	4	3	0	15	12	11	8	7	4	3	0
1	1	1	1	1	0	1	1	1		op1	Rn		RdLo		RdHi		op2		Rm	

Table C3-30

Decode fields		Instruction page
op1	op2	
000	!= 0000	Unallocated.
000	0000	SMULL, SMULLS
001	!= 1111	Unallocated.
001	1111	SDIV

Decode fields		Instruction page
op1	op2	
010	!= 0000	Unallocated.
010	0000	UMULL, UMULLS
011	!= 1111	Unallocated.
011	1111	UDIV
100	0000	SMLAL, SMLALS
100	0001	Unallocated.
100	001x	Unallocated.
100	01xx	Unallocated.
100	1000	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALBB variant
100	1001	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALBT variant
100	1010	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALTB variant
100	1011	SMLALBB, SMLALBT, SMLALTB, SMLALTT - SMLALTT variant
100	1100	SMLALD, SMLALDX - SMLALD variant
100	1101	SMLALD, SMLALDX - SMLALDX variant
100	111x	Unallocated.
101	0xxx	Unallocated.
101	10xx	Unallocated.
101	1100	SMLSLD, SMLSLDX - SMLSLD variant
101	1101	SMLSLD, SMLSLDX - SMLSLDX variant
101	111x	Unallocated.
110	0000	UMLAL, UMLALS
110	0001	Unallocated.
110	001x	Unallocated.
110	010x	Unallocated.
110	0110	UMAAL
110	0111	Unallocated.
110	1xxx	Unallocated.
111	xxxx	Unallocated.

C3.1.7 System register access, Advanced SIMD, and floating-point

This section describes the encoding of the System register access, Advanced SIMD, and floating-point group. The encodings in this section are decoded from 32-bit.

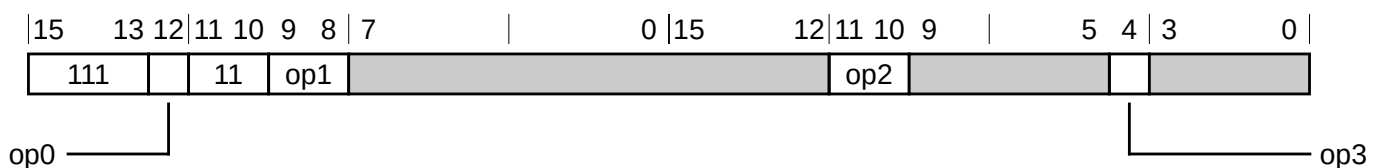


Table C3-31 Encoding table for the System register access, Advanced SIMD, and floating-point group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
x	0x	0x	x	Unallocated.
x	10	0x	x	Unallocated.
x	11	xx	x	Advanced SIMD data-processing
0	0x	1x	x	Advanced SIMD and System register load/store and 64-bit move
0	10	1x	1	Advanced SIMD and System register 32-bit move
0	10	10	0	Floating-point data-processing
0	10	11	0	Unallocated.
1	!= 11	1x	x	Additional Advanced SIMD and floating-point instructions

C3.1.8 Advanced SIMD data-processing

This section describes the encoding of the Advanced SIMD data-processing group. The encodings in this section are decoded from [System register access, Advanced SIMD, and floating-point](#).

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see [About the T32 Advanced SIMD and floating-point instructions and their encoding](#)

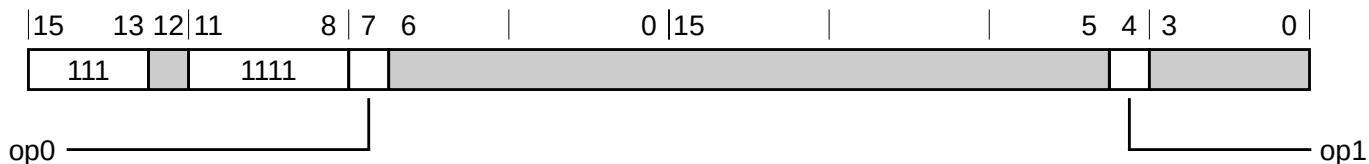


Table C3-32 Encoding table for the Advanced SIMD data-processing group

Decode fields		Decode group or instruction page
op0	op1	
0	x	Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

C3.1.8.1 Advanced SIMD three registers of the same length

This section describes the encoding of the Advanced SIMD three registers of the same length instruction class. The encodings in this section are decoded from [Advanced SIMD data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	8	7	6	5	4	3	0
1	1	1	U	1	1	1	1	0	D	size	Vn	Vd	opc	N	Q	M	o1	Vm					

Table C3-33

Decode fields					Instruction page	Feature
U	size	opc	Q	o1		
0	0x	1100	x	1	VFMA	-
0	0x	1101	x	0	VADD (floating-point)	-
0	0x	1101	x	1	VMLA (floating-point)	-
0	0x	1110	x	0	VCEQ (register) - T2	-
0	0x	1111	x	0	VMAX (floating-point)	-
0	0x	1111	x	1	VRECPS	-
x	xx	0000	x	0	VHADD	-
0	00	0001	x	1	VAND (register)	-
x	xx	0000	x	1	VQADD	-
x	xx	0001	x	0	VRHADD	-
0	00	1100	x	0	SHA1C	FEAT_SHA1
x	xx	0010	x	0	VHSUB	-
0	01	0001	x	1	VBIC (register)	-
x	xx	0010	x	1	VQSUB	-
x	xx	0011	x	0	VCGT (register) - T1	-
x	xx	0011	x	1	VCGE (register) - T1	-
0	01	1100	x	0	SHA1P	FEAT_SHA1
0	1x	1100	x	1	VFMS	-
0	1x	1101	x	0	VSUB (floating-point)	-
0	1x	1101	x	1	VMLS (floating-point)	-
0	1x	1110	x	0	Unallocated.	-
0	1x	1111	x	0	VMIN (floating-point)	-
0	1x	1111	x	1	VRSQRTS	-
x	xx	0100	x	0	VSHL (register)	-
0	xx	1000	x	0	VADD (integer)	-
0	10	0001	x	1	VORR (register)	-
0	xx	1000	x	1	VTST	-
x	xx	0100	x	1	VQSHL (register)	-
0	xx	1001	x	0	VMLA (integer)	-
x	xx	0101	x	0	VRSHL	-

Decode fields					Instruction page	Feature
U	size	opc	Q	o1		
x	xx	0101	x	1	VQRSHL	-
0	xx	1011	x	0	VQDMULH	-
0	10	1100	x	0	SHA1M	FEAT_SHA1
0	xx	1011	x	1	VPADD (integer)	-
x	xx	0110	x	0	VMAX (integer)	-
0	11	0001	x	1	VORN (register)	-
x	xx	0110	x	1	VMIN (integer)	-
x	xx	0111	x	0	VABD (integer)	-
x	xx	0111	x	1	VABA	-
0	11	1100	x	0	SHA1SU0	FEAT_SHA1
1	0x	1101	x	0	VPADD (floating-point)	-
1	0x	1101	x	1	VMUL (floating-point)	-
1	0x	1110	x	0	VCGE (register) - T2	-
1	0x	1110	x	1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111	x	1	VMAXNM	-
1	00	0001	x	1	VEOR	-
x	xx	1001	x	1	VMUL (integer and polynomial)	-
1	00	1100	x	0	SHA256H	FEAT_SHA256
x	xx	1010	0	0	VPMAX (integer)	-
1	01	0001	x	1	VBSL	-
x	xx	1010	0	1	VPMIN (integer)	-
x	xx	1010	1	x	Unallocated.	-
1	01	1100	x	0	SHA256H2	FEAT_SHA256
1	1x	1101	x	0	VABD (floating-point)	-
1	1x	1110	x	0	VCGT (register) - T2	-
1	1x	1110	x	1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111	x	1	VMINNM	-
1	xx	1000	x	0	VSUB (integer)	-
1	10	0001	x	1	VBIT	-
1	xx	1000	x	1	VCEQ (register) - T1	-
1	xx	1001	x	0	VMLS (integer)	-
1	xx	1011	x	0	VQRDMULH	-
1	10	1100	x	0	SHA256SU1	FEAT_SHA256
1	xx	1011	x	1	Unallocated.	-
1	11	0001	x	1	VBIF	-
1	xx	1100	x	1	Unallocated.	-
1	xx	1111	1	0	Unallocated.	-

C3.1.9 Advanced SIMD two registers, or three registers of different lengths

This section describes the encoding of the Advanced SIMD two registers, or three registers of different lengths group. The encodings in this section are decoded from [Advanced SIMD data-processing](#).

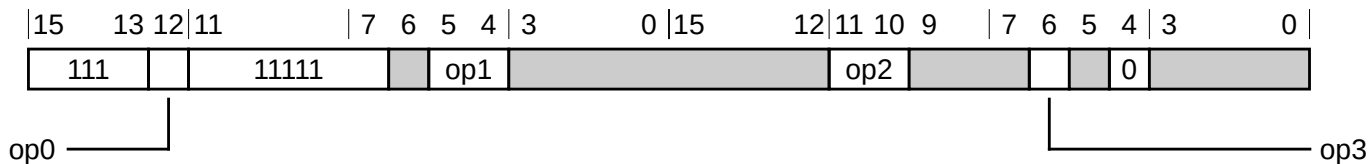


Table C3-34 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	11	xx	x	VEXT (byte elements)
1	11	0x	x	Advanced SIMD two registers misc
1	11	10	x	VTBL, VTBX
1	11	11	x	Advanced SIMD duplicate (scalar)
x	!= 11	xx	0	Advanced SIMD three registers of different lengths
x	!= 11	xx	1	Advanced SIMD two registers and a scalar

C3.1.9.1 Advanced SIMD two registers misc

This section describes the encoding of the Advanced SIMD two registers misc instruction class. The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

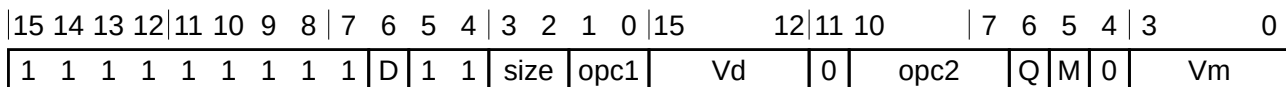


Table C3-35

Decode fields				Instruction page	Feature
size	opc1	opc2	Q		
xx	00	0000	x	VREV64	-
xx	00	0001	x	VREV32	-
xx	00	0010	x	VREV16	-
xx	00	0011	x	Unallocated.	-

Decode fields			Instruction page		Feature
size	opc1	opc2	Q		
xx	00	010x	x	VPADDL	-
xx	00	0110	0	AESE	FEAT_AES
xx	00	0110	1	AESD	FEAT_AES
xx	00	0111	0	AESMC	FEAT_AES
xx	00	0111	1	AESIMC	FEAT_AES
xx	00	1000	x	VCLS	-
00	10	0000	x	VSWP	-
xx	00	1001	x	VCLZ	-
xx	00	1010	x	VCNT	-
xx	00	1011	x	VMVN (register)	-
00	10	1100	1	Unallocated.	-
xx	00	110x	x	VPADAL	-
xx	00	1110	x	VQABS	-
xx	00	1111	x	VQNEG	-
xx	01	x000	x	VCGT (immediate #0)	-
xx	01	x001	x	VCGE (immediate #0)	-
xx	01	x010	x	VCEQ (immediate #0)	-
xx	01	x011	x	VCLE (immediate #0)	-
xx	01	x100	x	VCLT (immediate #0)	-
xx	01	x110	x	VABS	-
xx	01	x111	x	VNEG	-
xx	01	0101	1	SHA1H	FEAT_SHA1
01	10	1100	1	Unallocated.	-
xx	10	0001	x	VTRN	-
xx	10	0010	x	VUZP	-
xx	10	0011	x	VZIP	-
xx	10	0100	0	VMOVN	-
xx	10	0100	1	VQMOVN, VQMOVUN - Unsigned result variant	-
xx	10	0101	x	VQMOVN, VQMOVUN - Signed result variant	-
xx	10	0110	0	VSHLL	-
xx	10	0111	0	SHA1SU1	FEAT_SHA1
xx	10	0111	1	SHA256SU0	FEAT_SHA256
xx	10	1000	x	VRINTN (Advanced SIMD)	-
xx	10	1001	x	VRINTX (Advanced SIMD)	-
xx	10	1010	x	VRINTA (Advanced SIMD)	-
xx	10	1011	x	VRINTZ (Advanced SIMD)	-
10	10	1100	1	Unallocated.	-
xx	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Single-precision to half-precision variant	-

Decode fields			Instruction page		Feature
size	opc1	opc2	Q		
xx	10	1101	x	VRINTM (Advanced SIMD)	-
xx	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Half-precision to single-precision variant	-
xx	10	1110	1	Unallocated.	-
xx	10	1111	x	VRINTP (Advanced SIMD)	-
xx	11	000x	x	VCVTA (Advanced SIMD)	-
xx	11	001x	x	VCVTN (Advanced SIMD)	-
xx	11	010x	x	VCVTP (Advanced SIMD)	-
xx	11	011x	x	VCVTM (Advanced SIMD)	-
xx	11	10x0	x	VRECPE	-
xx	11	10x1	x	VRSQRTE	-
11	10	1100	1	Unallocated.	-
xx	11	11xx	x	VCVT (between floating-point and integer, Advanced SIMD)	-

C3.1.9.2 Advanced SIMD duplicate (scalar)

This section describes the encoding of the Advanced SIMD duplicate (scalar) instruction class. The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	7	6	5	4	3	0
1	1	1	1	1	1	1	1	1	D	1	1	imm4	Vd	1	1	opc	Q	M	0				Vm	

Table C3-36

Decode fields	Instruction page
opc	
000	VDUP (scalar)
001	Unallocated.
01x	Unallocated.
1xx	Unallocated.

C3.1.9.3 Advanced SIMD three registers of different lengths

This section describes the encoding of the Advanced SIMD three registers of different lengths instruction class. The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

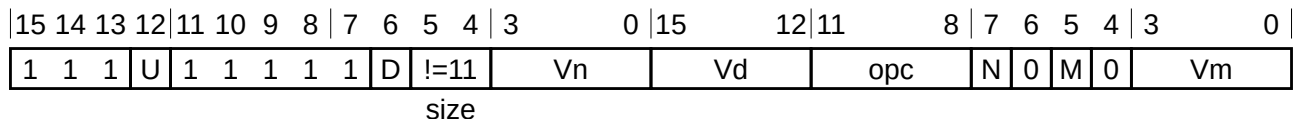


Table C3-37

Decode fields		Instruction page
U	opc	
x	0000	VADDL
x	0001	VADDW
x	0010	VSUBL
0	0100	VADDHN
x	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
x	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
x	0111	VABDL (integer)
x	1000	VMLAL (integer)
x	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
x	11x0	VMULL (integer and polynomial)
1	1001	Unallocated.
1	1011	Unallocated.
1	1101	Unallocated.
x	1111	Unallocated.

C3.1.9.4 Advanced SIMD two registers and a scalar

This section describes the encoding of the Advanced SIMD two registers and a scalar instruction class. The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

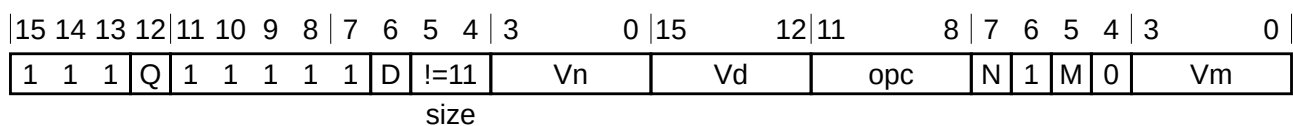


Table C3-38

Decode fields		Instruction page	Feature
Q	opc		
x	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
x	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
x	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
x	0110	VMLSL (by scalar)	-
x	100x	VMUL (by scalar)	-
1	0011	Unallocated.	-
x	1010	VMULL (by scalar)	-
1	0111	Unallocated.	-
x	1100	VQDMULH	-
x	1101	VQRDMULH	-
1	1x1x	Unallocated.	-

C3.1.10 Advanced SIMD shifts and immediate generation

This section describes the encoding of the Advanced SIMD shifts and immediate generation group. The encodings in this section are decoded from [Advanced SIMD data-processing](#).

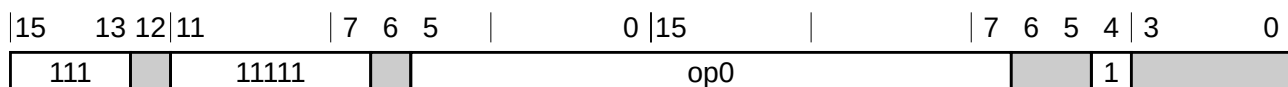


Table C3-39 Encoding table for the Advanced SIMD shifts and immediate generation group

Decode fields	Decode group or instruction page
op0	
000xxxxxxxxxxxx0	Advanced SIMD one register and modified immediate
!= 000xxxxxxxxxxxx0	Advanced SIMD two registers and shift amount

C3.1.10.1 Advanced SIMD one register and modified immediate

This section describes the encoding of the Advanced SIMD one register and modified immediate instruction class. The encodings in this section are decoded from [Advanced SIMD shifts and immediate generation](#).

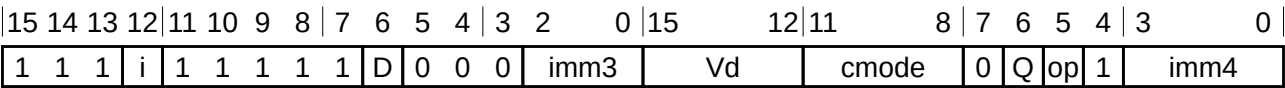
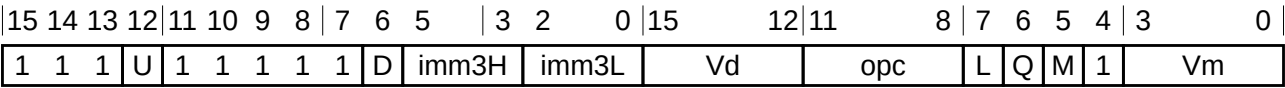


Table C3-40

Decode fields		Instruction page
cmode	op	
0xx0	0	VMOV (immediate) - T1
0xx0	1	VMVN (immediate) - T1
0xx1	0	VORR (immediate) - T1
0xx1	1	VBIC (immediate) - T1
10x0	0	VMOV (immediate) - T3
10x0	1	VMVN (immediate) - T2
10x1	0	VORR (immediate) - T2
10x1	1	VBIC (immediate) - T2
11xx	0	VMOV (immediate) - T4
110x	1	VMVN (immediate) - T3
1110	1	VMOV (immediate) - T5
1111	1	Unallocated.

C3.1.10.2 Advanced SIMD two registers and shift amount

This section describes the encoding of the Advanced SIMD two registers and shift amount instruction class. The encodings in this section are decoded from Advanced SIMD shifts and immediate generation.



This decode also imposes the constraint:

- !(imm3H == '000' && L == '0').

Table C3-41

Decode fields						Instruction page
U	imm3H	imm3L	opc	L	Q	
x	xxx	xxx	1xxx	1	x	Unallocated.
x	000	xxx	0000	1	0	VSHR 64-bit SIMD vector
x	000	xxx	0000	1	1	VSHR 128-bit SIMD vector
x	000	xxx	0001	1	0	VSRA 64-bit SIMD vector
x	000	xxx	0001	1	1	VSRA 128-bit SIMD vector
x	000	xxx	0010	1	0	VRSHR 64-bit SIMD vector
x	000	xxx	0010	1	1	VRSHR 128-bit SIMD vector
x	000	xxx	0011	1	0	VRSRA 64-bit SIMD vector
x	000	xxx	0011	1	1	VRSRA 128-bit SIMD vector
x	000	xxx	0111	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
x	000	xxx	0111	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
x	001	000	1010	0	0	VMOVL
x	001	!= 000	1010	0	0	VSHLL
x	010	000	1010	0	0	VMOVL
x	010	!= 000	1010	0	0	VSHLL
x	011	xxx	1010	0	0	VSHLL
x	100	000	1010	0	0	VMOVL
x	100	!= 000	1010	0	0	VSHLL
x	101	xxx	1010	0	0	VSHLL
x	11x	xxx	1010	0	0	VSHLL
x	!= 000	xxx	0000	x	0	VSHR 64-bit SIMD vector
x	!= 000	xxx	0000	x	1	VSHR 128-bit SIMD vector
x	!= 000	xxx	0001	x	0	VSRA 64-bit SIMD vector
x	!= 000	xxx	0001	x	1	VSRA 128-bit SIMD vector
x	!= 000	xxx	0010	x	0	VRSHR 64-bit SIMD vector
x	!= 000	xxx	0010	x	1	VRSHR 128-bit SIMD vector
x	!= 000	xxx	0011	x	0	VRSRA 64-bit SIMD vector
x	!= 000	xxx	0011	x	1	VRSRA 128-bit SIMD vector
x	!= 000	xxx	0111	x	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
x	!= 000	xxx	0111	x	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
x	!= 000	xxx	1001	0	0	VQSHRN, VQSHRUN — Signed result
x	!= 000	xxx	1001	0	1	VQRSHRN, VQRSHRUN — Signed result
x	!= 000	xxx	1010	0	1	Unallocated.
x	!= 000	xxx	1011	0	x	Unallocated.
x	!= 000	xxx	11xx	0	0	VCVT (between floating-point and fixed-point, Advanced SIMD) — 64-bit SIMD vector
x	!= 000	xxx	11xx	0	1	VCVT (between floating-point and fixed-point, Advanced SIMD) — 128-bit SIMD vector

Decode fields						Instruction page
U	imm3H	imm3L	opc	L	Q	
0	xxx	xxx	01x0	1	x	Unallocated.
0	000	xxx	0101	1	0	VSHL (immediate) — 64-bit SIMD vector
0	000	xxx	0101	1	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000	xxx	01x0	0	x	Unallocated.
0	!= 000	xxx	0101	x	0	VSHL (immediate) — 64-bit SIMD vector
0	!= 000	xxx	0101	x	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000	xxx	1000	0	0	VSHRN
0	!= 000	xxx	1000	0	1	VRSHRN
1	000	xxx	0100	1	0	VSRI — 64-bit SIMD vector
1	000	xxx	0100	1	1	VSRI — 128-bit SIMD vector
1	000	xxx	0101	1	0	VSLI — 64-bit SIMD vector
1	000	xxx	0101	1	1	VSLI — 128-bit SIMD vector
1	000	xxx	0110	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	000	xxx	0110	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000	xxx	0100	x	0	VSRI — 64-bit SIMD vector
1	!= 000	xxx	0100	x	1	VSRI — 128-bit SIMD vector
1	!= 000	xxx	0101	x	0	VSLI — 64-bit SIMD vector
1	!= 000	xxx	0101	x	1	VSLI — 128-bit SIMD vector
1	!= 000	xxx	0110	x	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	!= 000	xxx	0110	x	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000	xxx	1000	0	0	VQSHRN, VQSHRUN — Unsigned result
1	!= 000	xxx	1000	0	1	VQSHRN, VQSHRUN — Unsigned result

C3.1.11 Advanced SIMD and System register load/store and 64-bit move

This section describes the encoding of the Advanced SIMD and System register load/store and 64-bit move group. The encodings in this section are decoded from [System register access](#), [Advanced SIMD](#), and [floating-point](#).

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see [About the T32 Advanced SIMD and floating-point instructions and their encoding](#)

15	9	8	5	4	0	15	12	11	10	9	8	0
1110110				op0						1	op1	

Table C3-42 Encoding table for the Advanced SIMD and System register load/store and 64-bit move group

Decode fields		Decode group or instruction page
op0	op1	
00x0	0x	Advanced SIMD and floating-point 64-bit move
00x0	11	System register 64-bit move
!= 00x0	0x	Advanced SIMD and floating-point load/store
!= 00x0	11	System register Load/Store
xxxx	10	Unallocated.

C3.1.11.1 Advanced SIMD and floating-point 64-bit move

This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	0	0	0	D	0	op	Rt2	Rt	1	0	size	opc2	M	o3	Vm					

Table C3-43

Decode fields					Instruction page
D	op	size	opc2	o3	
0	x	xx	xx	x	Unallocated.
1	x	xx	xx	0	Unallocated.
1	x	0x	00	1	Unallocated.
1	x	xx	01	x	Unallocated.
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - From general-purpose registers variant
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - From general-purpose registers variant
1	x	xx	1x	x	Unallocated.
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - To general-purpose registers variant
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - To general-purpose registers variant

C3.1.11.2 System register 64-bit move

This section describes the encoding of the System register 64-bit move instruction class. The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

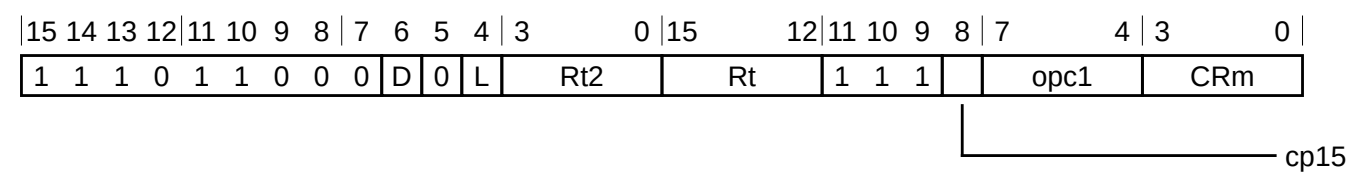


Table C3-44

Decode fields		Instruction page
D	L	
0	x	Unallocated.
1	0	MCRR
1	1	MRRC

C3.1.11.3 Advanced SIMD and floating-point load/store

This section describes the encoding of the Advanced SIMD and floating-point load/store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

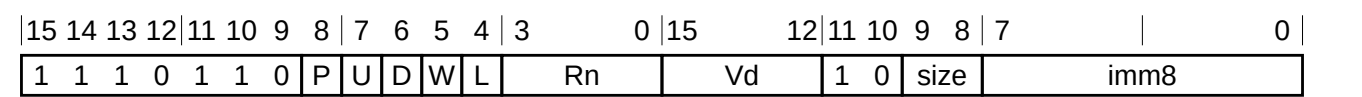


Table C3-45

Decode fields							Instruction page
P	U	W	L	Rn	size	imm8	
0	0	1	x	xxxx	xx	xxxxxxxx	Unallocated.
x	1	x	x	xxxx	0x	xxxxxxxx	Unallocated.
0	1	x	0	xxxx	10	xxxxxxxx	VSTM, VSTMDB, VSTMIA - Increment After variant
0	1	x	0	xxxx	11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA - Increment After variant
0	1	x	0	xxxx	11	xxxxxxxx1	FSTMDBX, FSTMIAX - Increment After variant
0	1	x	1	xxxx	10	xxxxxxxx	VLDM, VLDMDB, VLDMIA - Increment After variant
0	1	x	1	xxxx	11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA - Increment After variant
0	1	x	1	xxxx	11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - Increment After variant
1	x	0	0	xxxx	10	xxxxxxxx	VSTR - Single-precision scalar variant

Decode fields						Instruction page	
P	U	W	L	Rn	size	imm8	
1	x	0	0	xxxx	11	xxxxxxxx	VSTR - Double-precision scalar variant
1	x	0	1	!= 1111	01	xxxxxxxx	VLDR (immediate) - Half-precision scalar variant
1	x	0	1	!= 1111	10	xxxxxxxx	VLDR (immediate) - Single-precision scalar variant
1	x	0	1	!= 1111	11	xxxxxxxx	VLDR (immediate) - Double-precision scalar variant
1	0	1	x	xxxx	0x	xxxxxxxx	Unallocated.
1	0	1	0	xxxx	10	xxxxxxxx	VSTM, VSTMDB, VSTMIA - Decrement Before variant
1	0	1	0	xxxx	11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA - Decrement Before variant
1	0	1	0	xxxx	11	xxxxxxxx1	FSTMDBX, FSTMIA - Decrement Before variant
1	0	1	1	xxxx	10	xxxxxxxx	VLDM, VLDMDB, VLDMIA - Decrement Before variant
1	0	1	1	xxxx	11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA - Decrement Before variant
1	0	1	1	xxxx	11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIA) - Decrement Before variant
1	x	0	1	1111	10	xxxxxxxx	VLDR (literal) - Single-precision scalar variant
1	x	0	1	1111	11	xxxxxxxx	VLDR (literal) - Double-precision scalar variant
1	1	1	x	xxxx	xx	xxxxxxxx	Unallocated.

C3.1.11.4 System register Load/Store

This section describes the encoding of the System register Load/Store instruction class. The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

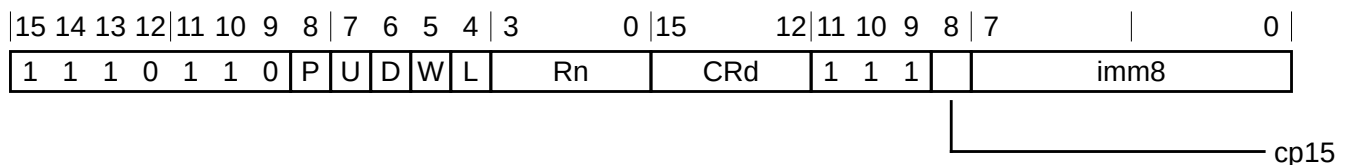


Table C3-46

Decode fields						Instruction page	
P:U:W	D	L	Rn	CRd	cp15		
!= 000	x	x	xxxx	!= 0101	0	Unallocated.	
!= 000	0	1	1111	0101	0	LDC (literal)	
!= 000	x	x	xxxx	xxxx	1	Unallocated.	
!= 000	1	x	xxxx	0101	0	Unallocated.	

Decode fields						Instruction page
P:U:W	D	L	Rn	CRd	cp15	
0x1	0	0	xxxx	0101	0	STC - Post-indexed variant
0x1	0	1	!= 1111	0101	0	LDC (immediate) - Post-indexed variant
010	0	0	xxxx	0101	0	STC - Unindexed variant
010	0	1	!= 1111	0101	0	LDC (immediate) - Unindexed variant
1x0	0	0	xxxx	0101	0	STC - Offset variant
1x0	0	1	!= 1111	0101	0	LDC (immediate) - Offset variant
1x1	0	0	xxxx	0101	0	STC - Pre-indexed variant
1x1	0	1	!= 1111	0101	0	LDC (immediate) - Pre-indexed variant

C3.1.12 Advanced SIMD and System register 32-bit move

This section describes the encoding of the Advanced SIMD and System register 32-bit move group. The encodings in this section are decoded from [System register access, Advanced SIMD, and floating-point](#).

15	8	7	5	4	0	15	12	11	10	8	7	5	4	3	0
11101110	op0							1	op1				1		

Table C3-47 Encoding table for the Advanced SIMD and System register 32-bit move group

Decode fields		Decode group or instruction page
op0	op1	
000	00x	Unallocated.
000	010	VMOV (between general-purpose register and single-precision)
001	010	Unallocated.
01x	010	Unallocated.
10x	010	Unallocated.
110	010	Unallocated.
111	010	Floating-point move special register
xxx	011	Advanced SIMD 8/16/32-bit element move/duplicate
xxx	10x	Unallocated.
xxx	11x	System register 32-bit move

C3.1.12.1 Floating-point move special register

This section describes the encoding of the Floating-point move special register instruction class. The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

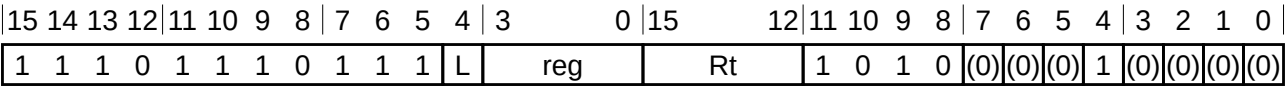


Table C3-48

Decode fields	Instruction page
L	
0	VMSR
1	VMRS

C3.1.12.2 Advanced SIMD 8/16/32-bit element move/duplicate

This section describes the encoding of the Advanced SIMD 8/16/32-bit element move/duplicate instruction class. The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

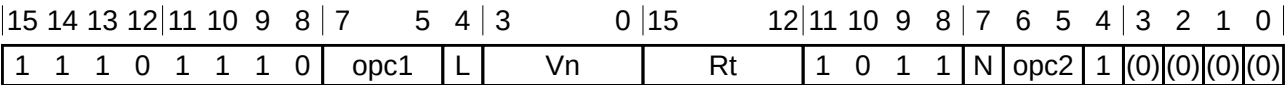


Table C3-49

Decode fields			Instruction page
opc1	L	opc2	
0xx	0	xx	VMOV (general-purpose register to scalar)
xxx	1	xx	VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	Unallocated.

C3.1.12.3 System register 32-bit move

This section describes the encoding of the System register 32-bit move instruction class. The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

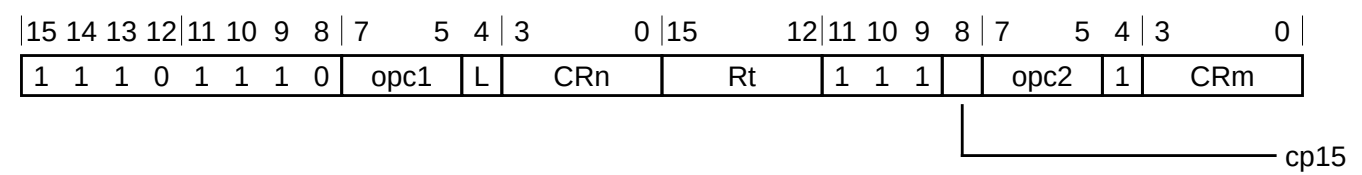


Table C3-50

Decode fields	Instruction page
L	
0	MCR
1	MRC

C3.1.13 Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from System register access, Advanced SIMD, and floating-point.
This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the T32 Advanced SIMD and floating-point instructions and their encoding



Table C3-51 Encoding table for the Floating-point data-processing group

Decode fields	Decode group or instruction page
op0	op1
1x11	1 Floating-point data-processing (two registers)
1x11	0 Floating-point move immediate
!= 1x11	x Floating-point data-processing (three registers)

C3.1.13.1 Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from Floating-point data-processing.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	1	D	1	1	o1	opc2		Vd		1	0	size	o3	1	M	0		Vm	

Table C3-52

Decode fields				Instruction page
o1	opc2	size	o3	
x	xxx	00	x	Unallocated.
0	000	01	x	Unallocated.
0	000	10	0	VMOV (register) - Single-precision scalar variant
0	000	10	1	VABS - Single-precision scalar variant
0	000	11	0	VMOV (register) - Double-precision scalar variant
0	000	11	1	VABS - Double-precision scalar variant
0	001	10	0	VNEG - Single-precision scalar variant
0	001	10	1	VSQRT - Single-precision scalar variant
0	001	11	0	VNEG - Double-precision scalar variant
0	001	11	1	VSQRT - Double-precision scalar variant
0	010	01	x	Unallocated.
0	010	10	0	VCVTB - Half-precision to single-precision variant
0	010	10	1	VCVTT - Half-precision to single-precision variant
0	010	11	0	VCVTB - Half-precision to double-precision variant
0	010	11	1	VCVTT - Half-precision to double-precision variant
0	011	01	x	Unallocated
0	011	10	0	VCVTB - Single-precision to half-precision variant
0	011	10	1	VCVTT - Single-precision to half-precision variant
0	011	11	0	VCVTB - Double-precision to half-precision variant
0	011	11	1	VCVTT - Double-precision to half-precision variant
0	100	10	0	VCMP
0	100	10	1	VCMPE
0	100	11	0	VCMP
0	100	11	1	VCMPE
0	101	10	0	VCMP
0	101	10	1	VCMPE
0	101	11	0	VCMP
0	101	11	1	VCMPE
0	110	10	0	VRINTR - Single-precision scalar variant
0	110	10	1	VRINTZ (floating-point) - Single-precision scalar variant

Decode fields				Instruction page
o1	opc2	size	o3	
0	110	11	0	VRINTR - Double-precision scalar variant
0	110	11	1	VRINTZ (floating-point) - Double-precision scalar variant
0	111	01	1	Unallocated.
0	111	10	0	VRINTX (floating-point) - Single-precision scalar variant
0	111	10	1	VCVT (between double-precision and single-precision) - Single-precision to double-precision variant
0	111	11	0	VRINTX (floating-point) - Double-precision scalar variant
0	111	11	1	VCVT (between double-precision and single-precision) - Double-precision to single-precision variant
1	000	10	x	VCVT (integer to floating-point, floating-point) - Single-precision scalar variant
1	000	11	x	VCVT (integer to floating-point, floating-point) - Double-precision scalar variant
1	001	01	x	Unallocated.
1	001	10	x	Unallocated.
1	001	11	x	Unallocated.
1	01x	10	x	VCVT (between floating-point and fixed-point, floating-point)
1	01x	11	x	VCVT (between floating-point and fixed-point, floating-point)
1	100	10	0	VCVTR
1	100	10	1	VCVT (floating-point to integer, floating-point)
1	100	11	0	VCVTR
1	100	11	1	VCVT (floating-point to integer, floating-point)
1	101	10	0	VCVTR
1	101	10	1	VCVT (floating-point to integer, floating-point)
1	101	11	0	VCVTR
1	101	11	1	VCVT (floating-point to integer, floating-point)
1	11x	10	x	VCVT (between floating-point and fixed-point, floating-point)
1	11x	11	x	VCVT (between floating-point and fixed-point, floating-point)

C3.1.13.2 Floating-point move immediate

This section describes the encoding of the Floating-point move immediate instruction class. The encodings in this section are decoded from [Floating-point data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H	Vd	1	0	size	(0)	0	(0)	0	imm4L				

Table C3-53

Decode fields size	Instruction page
0x	Unallocated.
10	VMOV (immediate) - Single-precision scalar variant
11	VMOV (immediate) - Double-precision scalar variant

C3.1.13.3 Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from [Floating-point data-processing](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	1	1	1	0	o0	D	o1	Vn			Vd		1	0	size	N	o2	M	0	Vm		

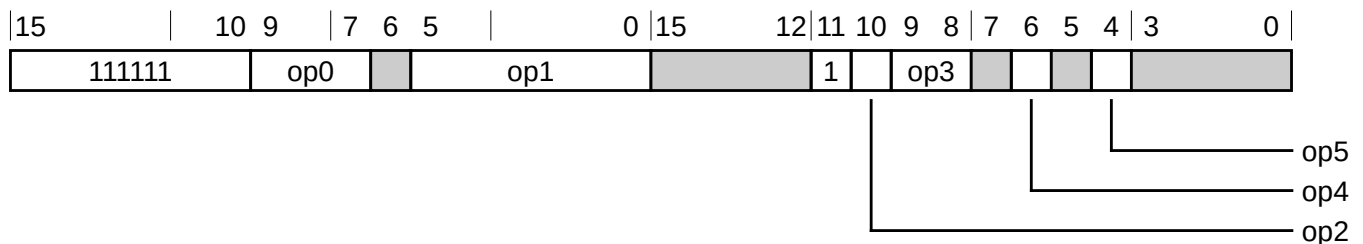
Table C3-54

Decode fields o0:o1	size	o2	Instruction page
!= 111	0x	x	Unallocated.
000	10	0	VMLA (floating-point) - Single-precision scalar variant
000	10	1	VMLS (floating-point) - Single-precision scalar variant
000	11	0	VMLA (floating-point) - Double-precision scalar variant
000	11	1	VMLS (floating-point) - Double-precision scalar variant
001	10	0	VNMLS - Single-precision scalar variant
001	10	1	VNMLA - Single-precision scalar variant
001	11	0	VNMLS - Double-precision scalar variant
001	11	1	VNMLA - Double-precision scalar variant
010	10	0	VMUL (floating-point) - Single-precision scalar variant
010	10	1	VNMUL - Single-precision scalar variant
010	11	0	VMUL (floating-point) - Double-precision scalar variant
010	11	1	VNMUL - Double-precision scalar variant
011	10	0	VADD (floating-point) - Single-precision scalar variant
011	10	1	VSUB (floating-point) - Single-precision scalar variant
011	11	0	VADD (floating-point) - Double-precision scalar variant
011	11	1	VSUB (floating-point) - Double-precision scalar variant
100	10	0	VDIV - Single-precision scalar variant
100	11	0	VDIV - Double-precision scalar variant

Decode fields			Instruction page
o0:o1	size	o2	
101	10	0	VFNMS - Single-precision scalar variant
101	10	1	VFNMA - Single-precision scalar variant
101	11	0	VFNMS - Double-precision scalar variant
101	11	1	VFNMA - Double-precision scalar variant
110	10	0	VFMA - Single-precision scalar variant
110	10	1	VFMS - Single-precision scalar variant
110	11	0	VFMA - Double-precision scalar variant
110	11	1	VFMS - Double-precision scalar variant

C3.1.14 Additional Advanced SIMD and floating-point instructions

This section describes the encoding of the Additional Advanced SIMD and floating-point instructions group. The encodings in this section are decoded from [System register access, Advanced SIMD, and floating-point](#).



This decode also imposes the constraint:

- $op0<2:1> \neq 11$.

Table C3-55 Encoding table for the Additional Advanced SIMD and floating-point instructions group

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0xx	xxxxxxx	x	0x	x	x	Unallocated.
100	xxxxxxx	0	!= 00	0	0	Floating-point conditional select
101	00xxxxx	0	!= 00	x	0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Unallocated.
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x	xxxxxxx	0	0x	x	x	Unallocated.

C3.1.14.1 Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

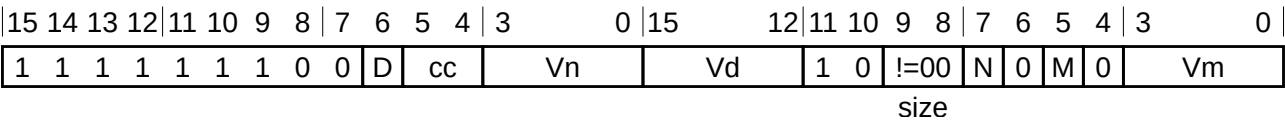


Table C3-56

Decode fields size	Instruction page
01	Unallocated.
10	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, single-precision scalar variant
11	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, double-precision scalar variant

C3.1.14.2 Floating-point minNum/maxNum

This section describes the encoding of the Floating-point minNum/maxNum instruction class. The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

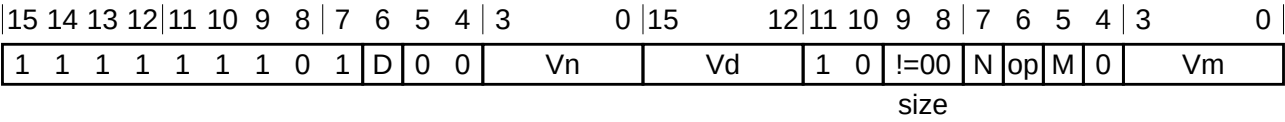


Table C3-57

Decode fields size	op	Instruction page
01	0	Unallocated.
10	0	VMAXNM - Single-precision scalar variant
10	1	VMINNM - Single-precision scalar variant
11	0	VMAXNM - Double-precision scalar variant
11	1	VMINNM - Double-precision scalar variant

C3.1.14.3 Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from [Additional Advanced SIMD and floating-point instructions](#).

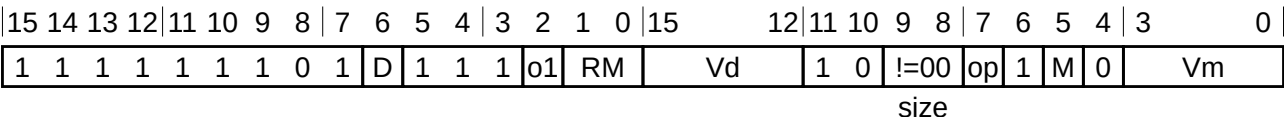


Table C3-58

Decode fields				Instruction page
o1	RM	size	op	
0	xx	!= 00	1	Unallocated.
0	xx	01	0	Unallocated.
0	00	10	0	VRINTA (floating-point) - Single-precision scalar variant
0	00	11	0	VRINTA (floating-point) - Double-precision scalar variant
0	01	10	0	VRINTN (floating-point) - Single-precision scalar variant
0	01	11	0	VRINTN (floating-point) - Double-precision scalar variant
0	10	10	0	VRINTP (floating-point) - Single-precision scalar variant
0	10	11	0	VRINTP (floating-point) - Double-precision scalar variant
0	11	10	0	VRINTM (floating-point) - Single-precision scalar variant
0	11	11	0	VRINTM (floating-point) - Double-precision scalar variant
1	00	10	x	VCVTA (floating-point) - Single-precision scalar variant
1	00	11	x	VCVTA (floating-point) - Double-precision scalar variant
1	01	10	x	VCVTN (floating-point) - Single-precision scalar variant
1	01	11	x	VCVTN (floating-point) - Double-precision scalar variant
1	10	10	x	VCVTP (floating-point) - Single-precision scalar variant
1	10	11	x	VCVTP (floating-point) - Double-precision scalar variant
1	11	10	x	VCVTM (floating-point) - Single-precision scalar variant
1	11	11	x	VCVTM (floating-point) - Double-precision scalar variant

C3.1.15 Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

This section describes the encoding of the Load/store dual, load/store exclusive, load-acquire/store-release, and table branch group. The encodings in this section are decoded from 32-bit.



This decode also imposes the constraint:

- op0<1> == 1.

Table C3-59 Encoding table for the Load/store dual, load/store exclusive, load-acquire/store-release, and table branch group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0010	x	xxxx	xxx	Load/store exclusive
0110	0	xxxx	000	Unallocated.
0110	1	xxxx	000	TBB, TBH
0110	x	xxxx	01x	Load/store exclusive byte/half/dual
0110	x	xxxx	1xx	Load-acquire / Store-release
0x11	x	!= 1111	xxx	Load/store dual (immediate, post-indexed)
1x10	x	!= 1111	xxx	Load/store dual (immediate)
1x11	x	!= 1111	xxx	Load/store dual (immediate, pre-indexed)
!= 0xx0	x	1111	xxx	LDRD (literal)

C3.1.15.1 Load/store exclusive

This section describes the encoding of the Load/store exclusive instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

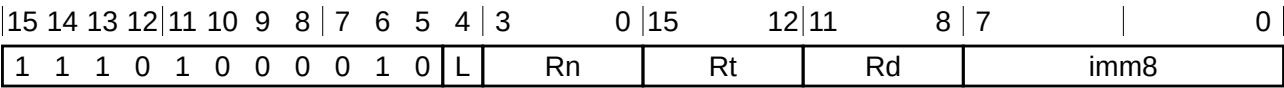


Table C3-60

Decode fields	Instruction page
L	
0	STREX
1	LDREX

C3.1.15.2 Load/store exclusive byte/half/dual

This section describes the encoding of the Load/store exclusive byte/half/dual instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

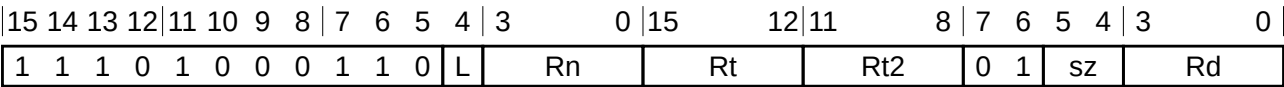


Table C3-61

Decode fields		Instruction page
L	sz	
0	00	STREXB
0	01	STREXH
0	10	Unallocated.
0	11	STREXD
1	00	LDREXB
1	01	LDREXH
1	10	Unallocated.
1	11	LDREXD

C3.1.15.3 Load-acquire / Store-release

This section describes the encoding of the Load-acquire / Store-release instruction class. The encodings in this section are decoded from [Load/store dual](#), [load/store exclusive](#), [load-acquire/store-release](#), and [table branch](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	8	7	6	5	4	3	0
1	1	1	0	1	0	0	0	1	1	0	L	Rn	Rt	Rt2	1	op	sz	Rd					

Table C3-62

Decode fields			Instruction page
L	op	sz	
0	0	00	STLB
0	0	01	STLH
0	0	10	STL
0	0	11	Unallocated.
0	1	00	STLEXB
0	1	01	STLEXH
0	1	10	STLEX
0	1	11	STLEXD
1	0	00	LDAB
1	0	01	LDAH
1	0	10	LDA
1	0	11	Unallocated.

Decode fields			Instruction page
L	op	sz	
1	1	00	LDAEXB
1	1	01	LDAEXH
1	1	10	LDAEX
1	1	11	LDAEXD

C3.1.15.4 Load/store dual (immediate, post-indexed)

This section describes the encoding of the Load/store dual (immediate, post-indexed) instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	8	7	0
1	1	1	0	1	0	0	0	U	1	1	L	!=1111	Rt	Rt2	imm8				
Rn																			

Table C3-63

Decode fields			Instruction page
L			
0			STRD (immediate)
1			LDRD (immediate)

C3.1.15.5 Load/store dual (immediate)

This section describes the encoding of the Load/store dual (immediate) instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	8	7	0
1	1	1	0	1	0	0	1	U	1	0	L	!=1111	Rt	Rt2	imm8				
Rn																			

Table C3-64

Decode fields			Instruction page
L			
0			STRD (immediate)

Decode fields	Instruction page
L	
1	LDRD (immediate)

C3.1.15.6 Load/store dual (immediate, pre-indexed)

This section describes the encoding of the Load/store dual (immediate, pre-indexed) instruction class. The encodings in this section are decoded from Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

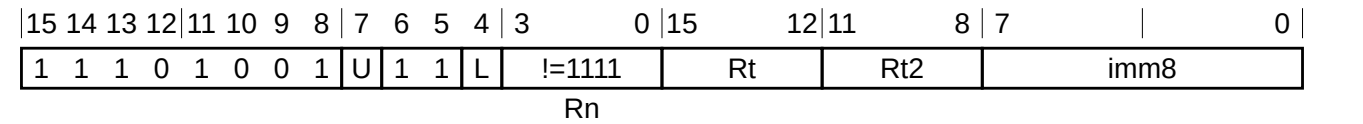


Table C3-65

Decode fields	Instruction page
L	
0	STRD (immediate)
1	LDRD (immediate)

C3.1.16 Branches and miscellaneous control

This section describes the encoding of the Branches and miscellaneous control group. The encodings in this section are decoded from 32-bit.

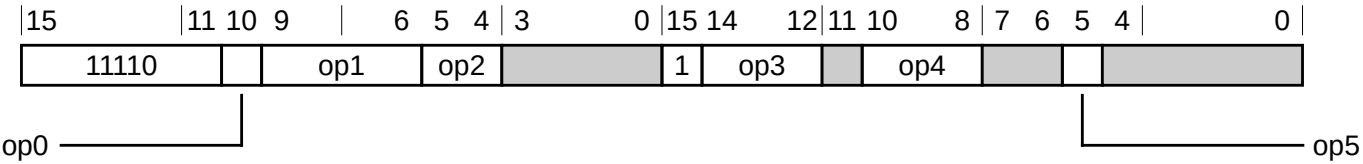


Table C3-66 Encoding table for the Branches and miscellaneous control group

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0	1110	0x	0x0	xxx	0	MSR (register)
0	1110	0x	0x0	xxx	1	MSR (Banked register)

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0	1110	10	0x0	000	x	Hints
0	1110	10	0x0	!= 000	x	Change processor state
0	1110	11	0x0	xxx	x	Miscellaneous system
0	1111	00	0x0	xxx	x	BXJ
0	1111	01	0x0	xxx	x	Exception return
0	1111	1x	0x0	xxx	0	MRS
0	1111	1x	0x0	xxx	1	MRS (Banked register)
1	1110	00	000	xxx	x	DCPS
1	1110	00	010	xxx	x	Unallocated.
1	1110	!= 00	0x0	xxx	x	Unallocated.
1	1111	0x	0x0	xxx	x	Unallocated.
1	1111	1x	0x0	xxx	x	Exception generation
x	!= 111x	x	0x0	xxx	x	B - T3 variant
x	xxxx	x	0x1	xxx	x	B - T4 variant
x	xxxx	x	1x0	xxx	x	BL, BLX (immediate) - T2 variant
x	xxxx	x	1x1	xxx	x	BL, BLX (immediate) - T1 variant

C3.1.16.1 Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7		4	3		0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0		hint		option		

Table C3-67

Decode fields		Instruction page
hint	option	
x100	xxxx	Reserved hint, behaves as NOP.
x101	xxxx	Reserved hint, behaves as NOP.
0x1x	xxxx	Reserved hint, behaves as NOP.
0000	0000	NOP
0000	0001	YIELD
0000	0010	WFE
0000	0011	WFI

Decode fields		Instruction page
hint	option	
0000	0100	SEV
0000	0101	SEVL
0000	011x	Reserved hint, behaves as NOP.
0000	1xxx	Reserved hint, behaves as NOP.
0001	xxxx	Reserved hint, behaves as NOP.
10xx	xxxx	Reserved hint, behaves as NOP.
1110	xxxx	Reserved hint, behaves as NOP.
1111	xxxx	DBG

C3.1.16.2 Change processor state

This section describes the encoding of the Change processor state instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod	M	A	I	F	mode		

Table C3-68

Decode fields		Instruction page
imod	M	
00	1	CPS, CPSID, CPSIE - Change mode variant
01	x	Unallocated.
10	x	CPS, CPSID, CPSIE - Interrupt enable and change mode variant
11	x	CPS, CPSID, CPSIE - Interrupt disable and change mode variant

C3.1.16.3 Miscellaneous system

This section describes the encoding of the Miscellaneous system instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7		4	3		0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	opc		option			

Table C3-69

Decode fields		Instruction page
opc	option	
000x	xxxx	Unallocated.
0010	xxxx	CLREX
0011	xxxx	Unallocated.
0100	!= 0x00	DSB
0100	0000	Unallocated.
0100	0100	Unallocated.
0101	xxxx	DMB
0110	xxxx	ISB
0111	xxxx	Unallocated.
1xxx	xxxx	Unallocated.

C3.1.16.4 Exception return

This section describes the encoding of the Exception return instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).

15 14 13 12 11 10 9 8 7 6 5 4 3	0	15 14 13 12 11 10 9 8 7	0
1 1 1 1 0 0 1 1 1 1 0 1	Rn	1 0 (0) 0 (1)(1)(1)(1)	imm8

Table C3-70

Decode fields	Instruction page
Rn:imm8	
!= 111000000000	SUB, SUBS (immediate)
111000000000	ERET

C3.1.16.5 DCPS

This section describes the encoding of the DCPS instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).

15 14 13 12 11 10 9 8 7 6 5 4 3	0	15 14 13 12 11		2 1 0
1 1 1 1 0 1 1 1 1 0 0 0	imm4	1 0 0 0	imm10	opt

Table C3-71

Decode fields			Instruction page
imm4	imm10	opt	
!= 1111	xxxxxxxxxx	xx	Unallocated.
1111	!= 0000000000	xx	Unallocated.
1111	0000000000	00	Unallocated.
1111	0000000000	01	DCPS1
1111	0000000000	10	DCPS2
1111	0000000000	11	DCPS3

C3.1.16.6 Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).

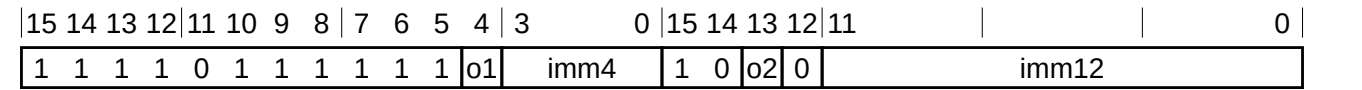


Table C3-72

Decode fields		Instruction page
o1	o2	
0	0	HVC
0	1	Unallocated.
1	0	SMC
1	1	UDF

C3.1.17 Data-processing (plain binary immediate)

This section describes the encoding of the Data-processing (plain binary immediate) group. The encodings in this section are decoded from [32-bit](#).

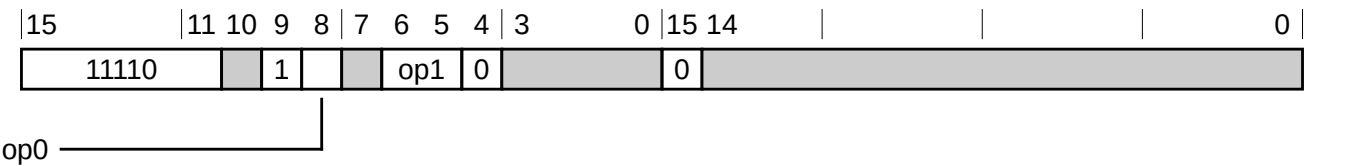


Table C3-73 Encoding table for the Data-processing (plain binary immediate) group

Decode fields		Decode group or instruction page
op0	op1	
0	0x	Data-processing (simple immediate)
0	10	Move Wide (16-bit immediate)
0	11	Unallocated.
1	xx	Saturate, Bitfield

C3.1.17.1 Data-processing (simple immediate)

This section describes the encoding of the Data-processing (simple immediate) instruction class. The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	14	12	11	8	7	0
1	1	1	1	0	i	1	0	o1	0	o2	0	Rn	0	imm3	Rd	imm8				

Table C3-74

Decode fields			Instruction page
o1	o2	Rn	
0	0	!= 11x1	ADD, ADDS (immediate)
0	0	1101	ADD, ADDS (SP plus immediate)
0	0	1111	ADR - T3
0	1	xxxx	Unallocated.
1	0	xxxx	Unallocated.
1	1	!= 11x1	SUB, SUBS (immediate)
1	1	1101	SUB, SUBS (SP minus immediate)
1	1	1111	ADR - T2

C3.1.17.2 Move Wide (16-bit immediate)

This section describes the encoding of the Move Wide (16-bit immediate) instruction class. The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	14	12	11	8	7	0
1	1	1	1	0	i	1	0	o1	1	0	0	imm4	0	imm3	Rd	imm8				

Table C3-75

Decode fields	Instruction page
o1	
0	MOV, MOVS (immediate)
1	MOVT

C3.1.17.3 Saturate, Bitfield

This section describes the encoding of the Saturate, Bitfield instruction class. The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).

15	14	13	12	11	10	9	8	7	5	4	3	0	15	14	12	11	8	7	6	5	4	0
1	1	1	1	0	(0)	1	1	op1	0	Rn	0	imm3	Rd	imm2	(0)	widthm1						

Table C3-76

Decode fields			Instruction page
op1	Rn	imm3:imm2	
000	xxxx	xxxxxx	SSAT - Logical shift left variant
001	xxxx	!= 00000	SSAT - Arithmetic shift right variant
001	xxxx	00000	SSAT16
010	xxxx	xxxxxx	SBFX
011	!= 1111	xxxxxx	BFI
011	1111	xxxxxx	BFC
100	xxxx	xxxxxx	USAT - Logical shift left variant
101	xxxx	!= 00000	USAT - Arithmetic shift right variant
101	xxxx	00000	USAT16
110	xxxx	xxxxxx	UBFX
111	xxxx	xxxxxx	Unallocated.

C3.1.18 Advanced SIMD element or structure load/store

This section describes the encoding of the Advanced SIMD element or structure load/store group. The encodings in this section are decoded from [32-bit](#).

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see [About the T32 Advanced SIMD and floating-point instructions and their encoding](#)

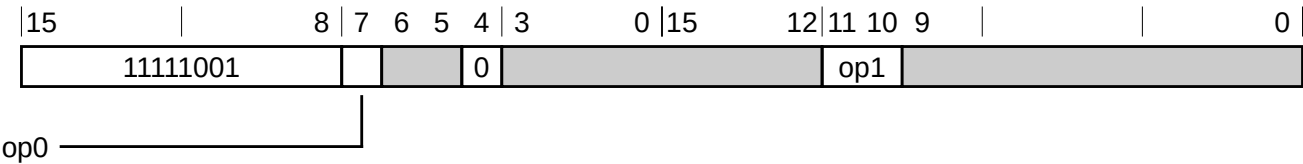


Table C3-77 Encoding table for the Advanced SIMD element or structure load/store group

Decode fields		Decode group or instruction page
op0	op1	
0	xx	Advanced SIMD load/store multiple structures
1	11	Advanced SIMD load single structure to all lanes
1	!= 11	Advanced SIMD load/store single structure to one lane

C3.1.18.1 Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from Advanced SIMD element or structure load/store.

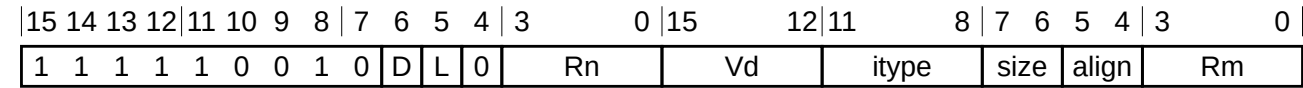


Table C3-78

Decode fields			Instruction page
L	itype	Rm	
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)

Decode fields			Instruction page
L	itype	Rm	
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
x	1011	xxxx	Unallocated.
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
x	11xx	xxxx	Unallocated.
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)

Decode fields			Instruction page
L	itype	Rm	
1	1010	1111	VLD1 (multiple single elements)

C3.1.18.2 Advanced SIMD load single structure to all lanes

This section describes the encoding of the Advanced SIMD load single structure to all lanes instruction class. The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	0	0	1	1	D	L	0	Rn	Vd	1	1	N	size	T	a	Rm					

Table C3-79

Decode fields				Instruction page
L	N	a	Rm	
0	x	x	xxxx	Unallocated.
1	00	x	!= 11x1	VLD1 (single element to all lanes)
1	00	x	1101	VLD1 (single element to all lanes)
1	00	x	1111	VLD1 (single element to all lanes)
1	01	x	!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01	x	1101	VLD2 (single 2-element structure to all lanes)
1	01	x	1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1	xxxx	Unallocated.
1	11	x	!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11	x	1101	VLD4 (single 4-element structure to all lanes)
1	11	x	1111	VLD4 (single 4-element structure to all lanes)

C3.1.18.3 Advanced SIMD load/store single structure to one lane

This section describes the encoding of the Advanced SIMD load/store single structure to one lane instruction class. The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	8	7	4	3	0
1	1	1	1	1	0	0	1	1	D	L	0	Rn	Vd	!=11	N	index_align	Rm						
size																							

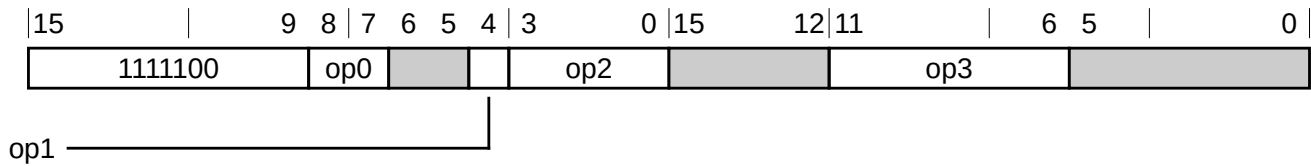
Table C3-80

Decode fields				Instruction page
L	size	N	Rm	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)

Decode fields				Instruction page
L	size	N	Rm	
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

C3.1.19 Load/store single

This section describes the encoding of the Load/store single group. The encodings in this section are decoded from 32-bit.



This decode also imposes the constraint:

- $op0 < 1 > : op1 \neq 10$.

Table C3-81 Encoding table for the Load/store single group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
00	x	!= 1111	000000	Load/store, unsigned (register offset)
00	x	!= 1111	000001	Unallocated.
00	x	!= 1111	00001x	Unallocated.
00	x	!= 1111	0001xx	Unallocated.
00	x	!= 1111	001xxx	Unallocated.
00	x	!= 1111	01xxxx	Unallocated.
00	x	!= 1111	10x0xx	Unallocated.
00	x	!= 1111	10x1xx	Load/store, unsigned (immediate, post-indexed)
00	x	!= 1111	1100xx	Load/store, unsigned (negative immediate)
00	x	!= 1111	1110xx	Load/store, unsigned (unprivileged)
00	x	!= 1111	11x1xx	Load/store, unsigned (immediate, pre-indexed)
01	x	!= 1111	xxxxxx	Load/store, unsigned (positive immediate)
0x	x	1111	xxxxxx	Load, unsigned (literal)
10	1	!= 1111	000000	Load/store, signed (register offset)
10	1	!= 1111	000001	Unallocated.
10	1	!= 1111	00001x	Unallocated.
10	1	!= 1111	0001xx	Unallocated.
10	1	!= 1111	001xxx	Unallocated.
10	1	!= 1111	01xxxx	Unallocated.
10	1	!= 1111	10x0xx	Unallocated.
10	1	!= 1111	10x1xx	Load/store, signed (immediate, post-indexed)
10	1	!= 1111	1100xx	Load/store, signed (negative immediate)
10	1	!= 1111	1110xx	Load/store, signed (unprivileged)

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
10	1	!= 1111	11x1xx	Load/store, signed (immediate, pre-indexed)
11	1	!= 1111	xxxxxxx	Load/store, signed (positive immediate)
1x	1	1111	xxxxxxx	Load, signed (literal)

C3.1.19.1 Load/store, unsigned (register offset)

This section describes the encoding of the Load/store, unsigned (register offset) instruction class. The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	0	0	0	0	size	L	!=1111	Rt	0	0	0	0	0	0	0	imm2	Rm				

Rn

Table C3-82

Decode fields			Instruction page
size	L	Rt	
00	0	xxxx	STRB (register)
00	1	!= 1111	LDRB (register)
00	1	1111	PLD, PLDW (register) - Preload read variant
01	0	xxxx	STRH (register)
01	1	!= 1111	LDRH (register)
01	1	1111	PLD, PLDW (register) - Preload write variant
10	0	xxxx	STR (register)
10	1	xxxx	LDR (register)
11	x	xxxx	Unallocated.

C3.1.19.2 Load/store, unsigned (immediate, post-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, post-indexed) instruction class. The encodings in this section are decoded from [Load/store single](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	8	7				0
1	1	1	1	1	0	0	0	0	size	L	!=1111	Rt	1	0	U	1	imm8							

Rn

Table C3-83

Decode fields	Instruction page	
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11	x	Unallocated.

C3.1.19.3 Load/store, unsigned (negative immediate)

This section describes the encoding of the Load/store, unsigned (negative immediate) instruction class. The encodings in this section are decoded from [Load/store single](#).

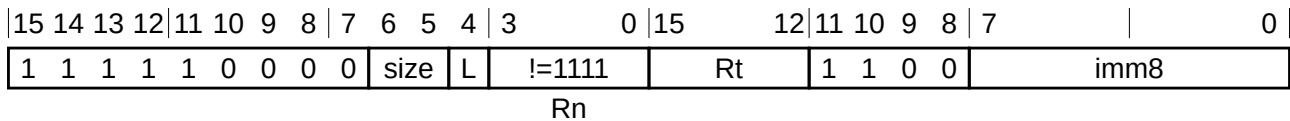


Table C3-84

Decode fields		Instruction page	
size	L	Rt	
00	0	xxxx	STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) - Preload read variant
01	0	xxxx	STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) - Preload write variant
10	0	xxxx	STR (immediate)
10	1	xxxx	LDR (immediate)
11	x	xxxx	Unallocated.

C3.1.19.4 Load/store, unsigned (unprivileged)

This section describes the encoding of the Load/store, unsigned (unprivileged) instruction class. The encodings in this section are decoded from [Load/store single](#).

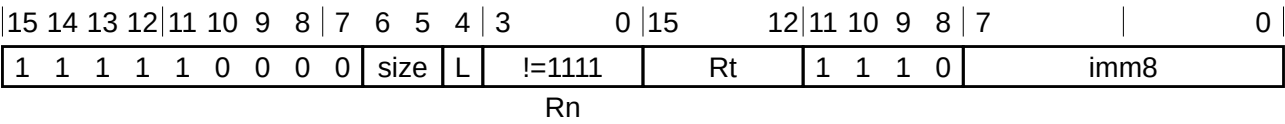


Table C3-85

Decode fields	Instruction page	
size	L	
00	0	STRBT
00	1	LDRBT
01	0	STRHT
01	1	LDRHT
10	0	STRT
10	1	LDRT
11	x	Unallocated.

C3.1.19.5 Load/store, unsigned (immediate, pre-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, pre-indexed) instruction class. The encodings in this section are decoded from [Load/store single](#).

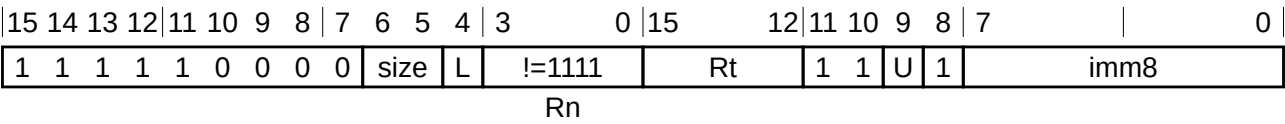


Table C3-86

Decode fields	Instruction page	
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11	x	Unallocated.

C3.1.19.6 Load/store, unsigned (positive immediate)

This section describes the encoding of the Load/store, unsigned (positive immediate) instruction class. The encodings in this section are decoded from [Load/store single](#).

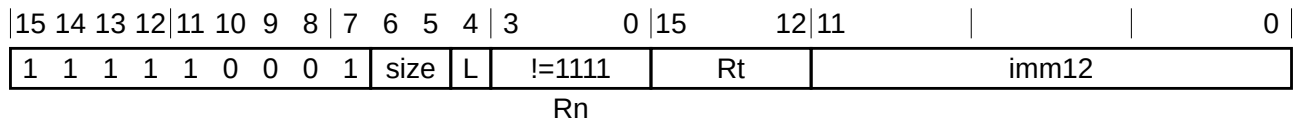


Table C3-87

Decode fields			Instruction page
size	L	Rt	
00	0	xxxx	STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) - Preload read variant
01	0	xxxx	STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) - Preload write variant
10	0	xxxx	STR (immediate)
10	1	xxxx	LDR (immediate)

C3.1.19.7 Load, unsigned (literal)

This section describes the encoding of the Load, unsigned (literal) instruction class. The encodings in this section are decoded from [Load/store single](#).

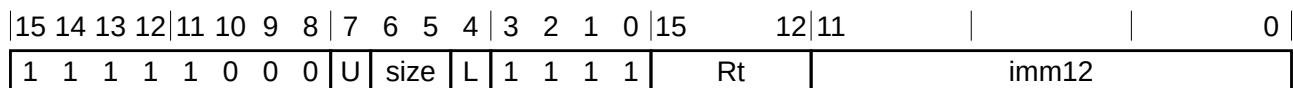


Table C3-88

Decode fields			Instruction page
size	L	Rt	
0x	1	1111	PLD (literal)
00	1	!= 1111	LDRB (literal)
01	1	!= 1111	LDRH (literal)

Decode fields			Instruction page
size	L	Rt	
10	1	xxxx	LDR (literal)
11	x	xxxx	Unallocated.

C3.1.19.8 Load/store, signed (register offset)

This section describes the encoding of the Load/store, signed (register offset) instruction class. The encodings in this section are decoded from [Load/store single](#).

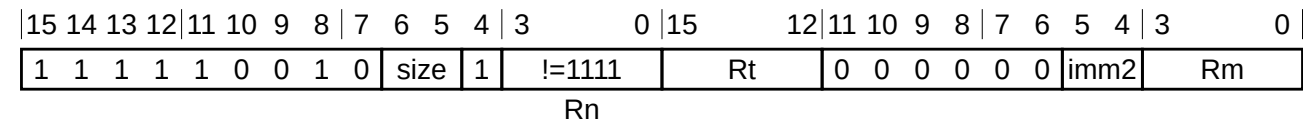


Table C3-89

Decode fields		Instruction page
size	Rt	
00	!= 1111	LDRSB (register)
00	1111	PLI (register)
01	!= 1111	LDRSH (register)
01	1111	Reserved hint, behaves as NOP.
1x	xxxx	Unallocated.

C3.1.19.9 Load/store, signed (immediate, post-indexed)

This section describes the encoding of the Load/store, signed (immediate, post-indexed) instruction class. The encodings in this section are decoded from [Load/store single](#).

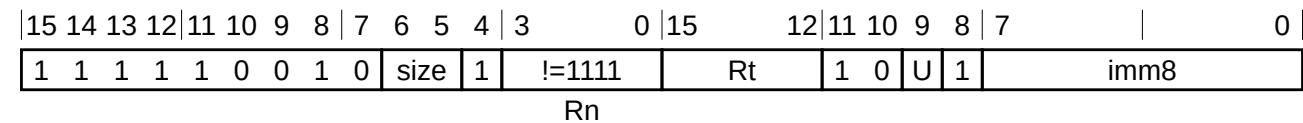


Table C3-90

Decode fields	Instruction page
size	
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	Unallocated.

C3.1.19.10 Load/store, signed (negative immediate)

This section describes the encoding of the Load/store, signed (negative immediate) instruction class. The encodings in this section are decoded from [Load/store single](#).

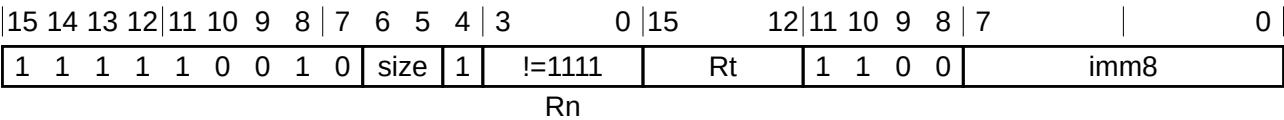


Table C3-91

Decode fields	Rt	Instruction page
size		
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP.
1x	xxxx	Unallocated.

C3.1.19.11 Load/store, signed (unprivileged)

This section describes the encoding of the Load/store, signed (unprivileged) instruction class. The encodings in this section are decoded from [Load/store single](#).

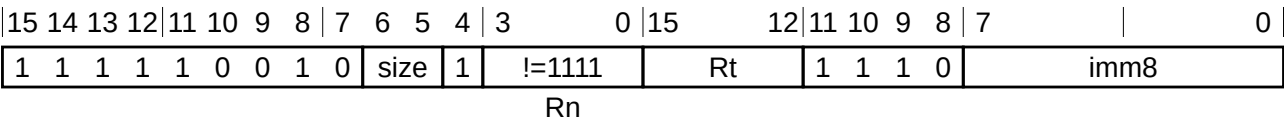


Table C3-92

Decode fields	Instruction page
size	
00	LDRSBT
01	LDRSHT
1x	Unallocated.

C3.1.19.12 Load/store, signed (immediate, pre-indexed)

This section describes the encoding of the Load/store, signed (immediate, pre-indexed) instruction class. The encodings in this section are decoded from [Load/store single](#).

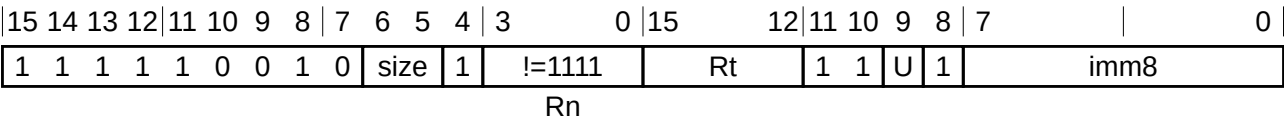


Table C3-93

Decode fields	Instruction page
size	
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	Unallocated.

C3.1.19.13 Load/store, signed (positive immediate)

This section describes the encoding of the Load/store, signed (positive immediate) instruction class. The encodings in this section are decoded from [Load/store single](#).

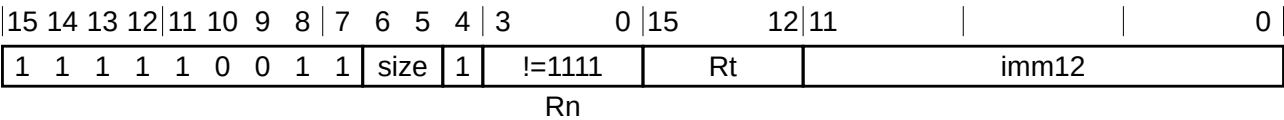


Table C3-94

Decode fields		Instruction page
size	Rt	
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP.

C3.1.19.14 Load, signed (literal)

This section describes the encoding of the Load, signed (literal) instruction class. The encodings in this section are decoded from [Load/store single](#).

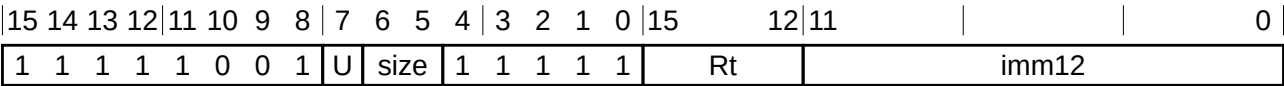


Table C3-95

Decode fields		Instruction page
size	Rt	
00	!= 1111	LDRSB (literal)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (literal)
01	1111	Reserved hint, behaves as NOP.
1x	xxxx	Unallocated.

C3.1.20 Data-processing (register)

This section describes the encoding of the Data-processing (register) group. The encodings in this section are decoded from [32-bit](#).

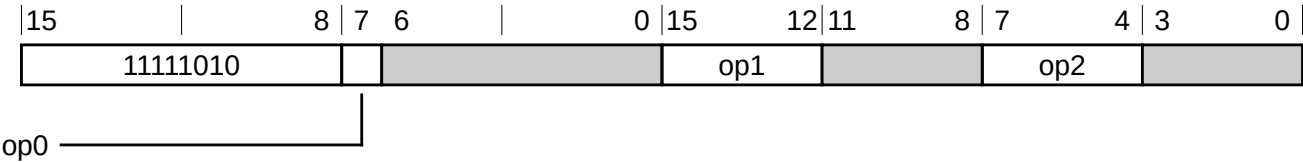


Table C3-96 Encoding table for the Data-processing (register) group

Decode fields			Decode group or instruction page
op0	op1	op2	
0	1111	0000	MOV, MOVS (register-shifted register) - Flag setting variant
0	1111	0001	Unallocated.
0	1111	001x	Unallocated.
0	1111	01xx	Unallocated.
0	1111	1xxx	Register extends
1	1111	0xxx	Parallel add-subtract
1	1111	10xx	Data-processing (two source registers)
1	1111	11xx	Unallocated.
x	!= 1111	xxxx	Unallocated.

C3.1.20.1 Register extends

This section describes the encoding of the Register extends instruction class. The encodings in this section are decoded from [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	0	15	14	13	12	11	8	7	6	5	4	3	0
1	1	1	1	1	0	1	0	0	op1	U		Rn		1	1	1	1		Rd	1	(0)	rotate		Rm	

Table C3-97

Decode fields			Instruction page
op1	U	Rn	
00	0	!= 1111	SXTAH
00	0	1111	SXTH
00	1	!= 1111	UXTAH
00	1	1111	UXTH
01	0	!= 1111	SXTAB16
01	0	1111	SXTB16
01	1	!= 1111	UXTAB16
01	1	1111	UXTB16
10	0	!= 1111	SXTAB
10	0	1111	SXTB
10	1	!= 1111	UXTAB
10	1	1111	UXTB
11	x	xxxx	Unallocated.

C3.1.20.2 Parallel add-subtract

This section describes the encoding of the Parallel add-subtract instruction class. The encodings in this section are decoded from [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	4	3	0	15	14	13	12	11	8	7	6	5	4	3	0
1	1	1	1	1	0	1	0	1	op1		Rn		1	1	1	1	Rd		0	U	H	S	Rm	

Table C3-98

Decode fields				Instruction page
op1	U	H	S	
000	0	0	0	SADD8
000	0	0	1	QADD8
000	0	1	0	SHADD8
000	0	1	1	Unallocated.
000	1	0	0	UADD8
000	1	0	1	UQADD8
000	1	1	0	UHADD8
000	1	1	1	Unallocated.
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	0	1	1	Unallocated.
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
001	1	1	1	Unallocated.
010	0	0	0	SASX
010	0	0	1	QASX
010	0	1	0	SHASX
010	0	1	1	Unallocated.
010	1	0	0	UASX
010	1	0	1	UQASX
010	1	1	0	UHASX
010	1	1	1	Unallocated.
100	0	0	0	SSUB8
100	0	0	1	QSUB8

Decode fields				Instruction page
op1	U	H	S	
100	0	1	0	SHSUB8
100	0	1	1	Unallocated.
100	1	0	0	USUB8
100	1	0	1	UQSUB8
100	1	1	0	UHSUB8
100	1	1	1	Unallocated.
101	0	0	0	SSUB16
101	0	0	1	QSUB16
101	0	1	0	SHSUB16
101	0	1	1	Unallocated.
101	1	0	0	USUB16
101	1	0	1	UQSUB16
101	1	1	0	UHSUB16
101	1	1	1	Unallocated.
110	0	0	0	SSAX
110	0	0	1	QSAX
110	0	1	0	SHSAX
110	0	1	1	Unallocated.
110	1	0	0	USAX
110	1	0	1	UQSAX
110	1	1	0	UHSAX
110	1	1	1	Unallocated.
111	x	x	x	Unallocated.

C3.1.20.3 Data-processing (two source registers)

This section describes the encoding of the Data-processing (two source registers) instruction class. The encodings in this section are decoded from [Data-processing \(register\)](#).

15	14	13	12	11	10	9	8	7	6	4	3	0	15	14	13	12	11	8	7	6	5	4	3	0
1	1	1	1	1	0	1	0	1	op1	Rn	1	1	1	1	Rd	1	0	op2	Rm					

Table C3-99

Decode fields		Instruction page	Feature
op1	op2		
000	00	QADD	-
000	01	QDADD	-
000	10	QSUB	-
000	11	QDSUB	-
001	00	REV	-
001	01	REV16	-
001	10	RBIT	-
001	11	REVSH	-
010	00	SEL	-
010	01	Unallocated.	-
010	1x	Unallocated.	-
011	00	CLZ	-
011	01	Unallocated.	-
011	1x	Unallocated.	-
100	00	CRC32 - CRC32B variant	FEAT_CRC32
100	01	CRC32 - CRC32H variant	FEAT_CRC32
100	10	CRC32 - CRC32W variant	FEAT_CRC32
100	11	CONSTRAINED UNPREDICTABLE	-
101	00	CRC32C - CRC32CB variant	FEAT_CRC32
101	01	CRC32C - CRC32CH variant	FEAT_CRC32
101	10	CRC32C - CRC32CW variant	FEAT_CRC32
101	11	CONSTRAINED UNPREDICTABLE	-
11x	xx	Unallocated.	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings](#).

C3.1.21 Multiply, multiply accumulate, and absolute difference

This section describes the encoding of the Multiply, multiply accumulate, and absolute difference group. The encodings in this section are decoded from [32-bit](#).

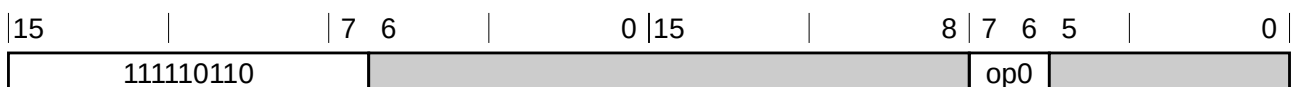


Table C3-100 Encoding table for the Multiply, multiply accumulate, and absolute difference group

Decode fields	Decode group or instruction page
op0	
00	Multiply and absolute difference
01	Unallocated.
1x	Unallocated.

C3.1.21.1 Multiply and absolute difference

This section describes the encoding of the Multiply and absolute difference instruction class. The encodings in this section are decoded from [Multiply, multiply accumulate, and absolute difference](#).

15	14	13	12	11	10	9	8	7	6	4	3	0	15	12	11	8	7	6	5	4	3	0
1	1	1	1	1	0	1	1	0	op1	Rn	Ra	Rd	0	0	op2	Rm						

Table C3-101

Decode fields	Instruction page		
op1	Ra	op2	
000	!= 1111	00	MLA, MLAS
000	xxxx	01	MLS
000	xxxx	1x	Unallocated.
000	1111	00	MUL, MULS
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT - SMLABB variant
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT - SMLABT variant
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT - SMLATB variant
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT - SMLATT variant
001	1111	00	SMULBB, SMULBT, SMULTB, SMULTT - SMULBB variant
001	1111	01	SMULBB, SMULBT, SMULTB, SMULTT - SMULBT variant
001	1111	10	SMULBB, SMULBT, SMULTB, SMULTT - SMULTB variant
001	1111	11	SMULBB, SMULBT, SMULTB, SMULTT - SMULTT variant
010	!= 1111	00	SMLAD, SMLADX - SMLAD variant
010	!= 1111	01	SMLAD, SMLADX - SMLADX variant
010	xxxx	1x	Unallocated.
010	1111	00	SMUAD, SMUADX - SMUAD variant
010	1111	01	SMUAD, SMUADX - SMUADX variant
011	!= 1111	00	SMLAWB, SMLAWT - SMLAWB variant
011	!= 1111	01	SMLAWB, SMLAWT - SMLAWT variant

Decode fields			Instruction page
op1	Ra	op2	
011	xxxx	1x	Unallocated.
011	1111	00	SMULWB, SMULWT - SMULWB variant
011	1111	01	SMULWB, SMULWT - SMULWT variant
100	!= 1111	00	SMLSD, SMLSDX - SMLSD variant
100	!= 1111	01	SMLSD, SMLSDX - SMLSDX variant
100	xxxx	1x	Unallocated.
100	1111	00	SMUSD, SMUSDX - SMUSD variant
100	1111	01	SMUSD, SMUSDX - SMUSDX variant
101	!= 1111	00	SMMLA, SMMLAR - SMMLA variant
101	!= 1111	01	SMMLA, SMMLAR - SMMLAR variant
101	xxxx	1x	Unallocated.
101	1111	00	SMMUL, SMMULR - SMMUL variant
101	1111	01	SMMUL, SMMULR - SMMULR variant
110	xxxx	00	SMMLS, SMMLSR - SMMLS variant
110	xxxx	01	SMMLS, SMMLSR - SMMLSR variant
110	xxxx	1x	Unallocated.
111	!= 1111	00	USADA8
111	xxxx	01	Unallocated.
111	xxxx	1x	Unallocated.
111	1111	00	USAD8

C3.2 About the T32 Advanced SIMD and floating-point instructions and their encoding

The Advanced SIMD and floating-point instructions are common to the T32 and A32 instruction sets. These instructions perform Advanced SIMD and floating-point operations on a common register file, the SIMD&FP register file. This means:

- In general, the instructions that load or store registers in this file, or move data between general-purpose registers and this register file, are common to the Advanced SIMD and floating-point instructions.
- There are distinct Advanced SIMD data-processing instructions and floating-point data-processing instructions.

All T32 Advanced SIMD and floating-point instructions have 32-bit encodings. Different groups of these instructions are decoded from different points in the 32-bit T32 instruction decode structure. [Table C3-102](#) shows these instruction groups, and where each group is decoded from the overall T32 decode structure:

Table C3-102 Advanced SIMD and floating-point instructions in the T32 decode structure

Advanced SIMD and floating-point instruction group	T32 decode is from
Advanced SIMD and System register load/store and 64-bit move	System register access, Advanced SIMD, and floating-point
Floating-point data-processing	System register access, Advanced SIMD, and floating-point
Advanced SIMD and System register 32-bit move	System register access, Advanced SIMD, and floating-point
Advanced SIMD data-processing	System register access, Advanced SIMD, and floating-point
Advanced SIMD element or structure load/store	32-bit

Chapter C4

A32 Instruction Set Encoding

This chapter describes the encoding of the A32 instruction set. It contains the following sections:

- [A32 instruction set encoding](#)
- [About the A32 Advanced SIMD and floating-point instructions and their encoding](#)

In this chapter:

- In the decode tables, an entry of $\times$ for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.

C4.1 A32 instruction set encoding

The A32 instruction stream is a sequence of 32-bit word-aligned words. Each instruction is a single word in the stream.

Most A32 instructions can be conditional, with a condition determined by bits[31:28] of the instruction, the `cond` field. For more information see [The Condition code field in A32 instruction encodings](#). This applies to all instructions except those with the `cond` field equal to `0b1111`.

The behavior of an attempt to execute an unallocated instruction is described in [UNDEFINED, UNPREDICTABLE, and CONSTRAINED UNPREDICTABLE instruction set space](#).

For more information on A32 instruction encodings see [About the T32 and A32 Instruction Descriptions](#).

The A32 instruction encoding is:

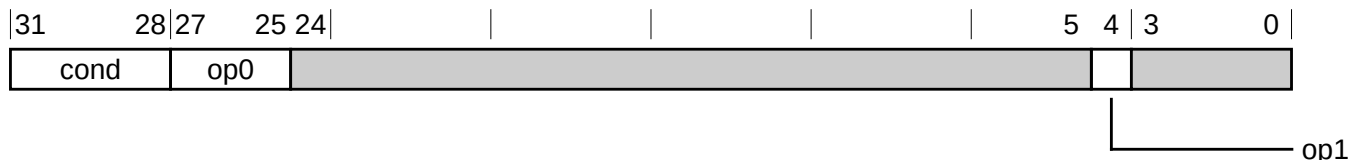


Table C4-1 Main encoding table for the A32 instruction set

Decode fields			Decode group or instruction page
cond	op0	op1	
!= 1111	00x	x	Data-processing and miscellaneous instructions
!= 1111	010	x	Load/Store Word, Unsigned Byte (immediate, literal)
!= 1111	011	0	Load/Store Word, Unsigned Byte (register)
!= 1111	011	1	Media instructions
xxxx	10x	x	Branch, branch with link, and block data transfer
xxxx	11x	x	System register access, Advanced SIMD, floating-point, and Supervisor call
1111	0xx	x	Unconditional instructions

C4.1.1 Data-processing and miscellaneous instructions

This section describes the encoding of the Data-processing and miscellaneous instructions group. The encodings in this section are decoded from [A32 instruction set encoding](#).



Table C4-2 Encoding table for the Data-processing and miscellaneous instructions group

Decode fields					Decode group or instruction page
op0	op1	op2	op3	op4	
0	xxxxxx	1	!= 00	1	Extra load/store
0	0xxxxx	1	00	1	Multiply and Accumulate
0	1xxxxx	1	00	1	Synchronization primitives and Load-Acquire/Store-Release
0	10xx0	0	xx	x	Miscellaneous
0	10xx0	1	xx	0	Halfword Multiply and Accumulate
0	!= 10xx0	x	xx	0	Data-processing register (immediate shift)
0	!= 10xx0	0	xx	1	Data-processing register (register shift)
1	xxxxxx	x	xx	x	Data-processing immediate

C4.1.1.1 Multiply and Accumulate

This section describes the encoding of the Multiply and Accumulate instruction class. The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	28	27	26	25	24	23	21	20	19	16	15	12	11	8	7	6	5	4	3	0										
!=1111				0 0 0 0				opc		S	RdHi				RdLo				Rm				1 0 0 1				Rn			
cond																														

Table C4-3

Decode fields		Instruction page
opc	S	
000	x	MUL, MULS
001	x	MLA, MLAS
010	0	UMAAL
010	1	Unallocated.
011	0	MLS
011	1	Unallocated.
100	x	UMULL, UMULLS
101	x	UMLAL, UMLALS
110	x	SMULL, SMULLS
111	x	SMLAL, SMLALS

C4.1.1.2 Halfword Multiply and Accumulate

This section describes the encoding of the Halfword Multiply and Accumulate instruction class. The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

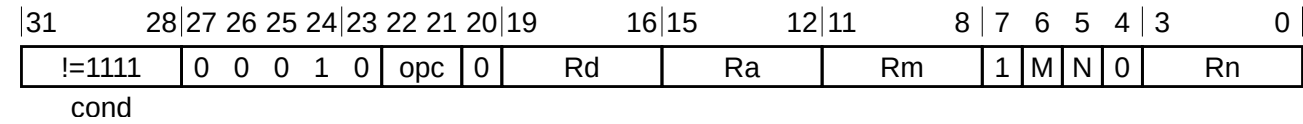


Table C4-4

Decode fields	Instruction page		
op0	M	N	
00	x	x	SMLABB, SMLABT, SMLATB, SMLATT
01	0	0	SMLAWB, SMLAWT - SMLAWB variant
01	0	1	SMULWB, SMULWT - SMULWB variant
01	1	0	SMLAWB, SMLAWT - SMLAWT variant
01	1	1	SMULWB, SMULWT - SMULWT variant
10	x	x	SMLALBB, SMLALBT, SMLALTB, SMLALTT
11	x	x	SMULBB, SMULBT, SMULTB, SMULTT

C4.1.2 Extra load/store

This section describes the encoding of the Extra load/store group. The encodings in this section are decoded from Data-processing and miscellaneous instructions.

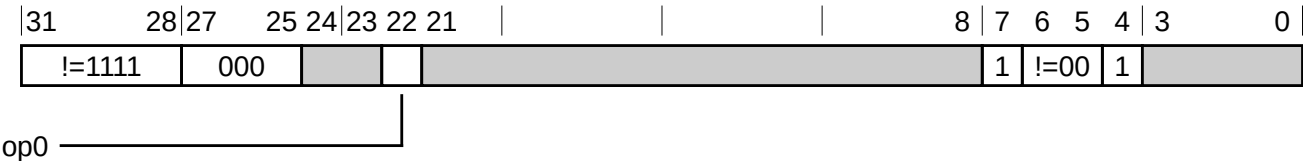


Table C4-5 Encoding table for the Extra load/store group

Decode fields	Decode group or instruction page
op0	
0	Load/Store Dual, Half, Signed Byte (register)
1	Load/Store Dual, Half, Signed Byte (immediate, literal)

C4.1.2.1 Load/Store Dual, Half, Signed Byte (register)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (register) instruction class. The encodings in this section are decoded from Extra load/store.

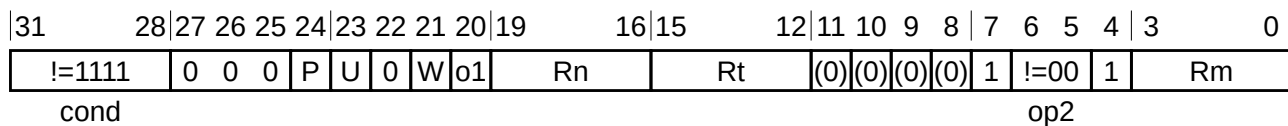


Table C4-6

Decode fields				Instruction page
P	W	o1	op2	
0	0	0	01	STRH (register) - Post-indexed variant
0	0	0	10	LDRD (register) - Post-indexed variant
0	0	0	11	STRD (register) - Post-indexed variant
0	0	1	01	LDRH (register) - Post-indexed variant
0	0	1	10	LDRSB (register) - Post-indexed variant
0	0	1	11	LDRSH (register) - Post-indexed variant
0	1	0	01	STRHT
0	1	0	10	Unallocated.
0	1	0	11	Unallocated.
0	1	1	01	LDRHT
0	1	1	10	LDRSBT
0	1	1	11	LDRSHT
1	x	0	01	STRH (register) - Pre-indexed variant
1	x	0	10	LDRD (register) - Pre-indexed variant
1	x	0	11	STRD (register) - Pre-indexed variant
1	x	1	01	LDRH (register) - Pre-indexed variant
1	x	1	10	LDRSB (register) - Pre-indexed variant
1	x	1	11	LDRSH (register) - Pre-indexed variant

C4.1.2.2 Load/Store Dual, Half, Signed Byte (immediate, literal)

This section describes the encoding of the Load/Store Dual, Half, Signed Byte (immediate, literal) instruction class. The encodings in this section are decoded from [Extra load/store](#).

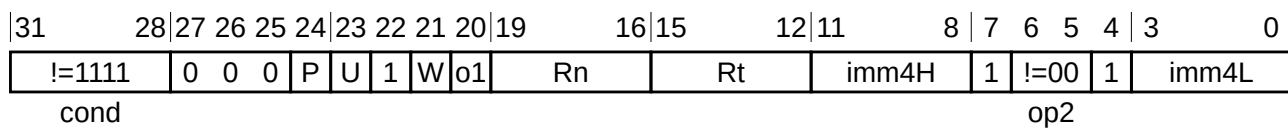


Table C4-7

Decode fields				Instruction page
P:W	o1	Rn	op2	
xx	0	1111	10	LDRD (literal)
!= 01	1	1111	01	LDRH (literal)
!= 01	1	1111	10	LDRSB (literal)
!= 01	1	1111	11	LDRSH (literal)
00	0	!= 1111	10	LDRD (immediate) - Post-indexed variant
00	0	xxxx	01	STRH (immediate) - Post-indexed variant
00	0	xxxx	11	STRD (immediate) - Post-indexed variant
00	1	!= 1111	01	LDRH (immediate) - Post-indexed variant
00	1	!= 1111	10	LDRSB (immediate) - Post-indexed variant
00	1	!= 1111	11	LDRSH (immediate) - Post-indexed variant
01	0	!= 1111	10	Unallocated.
01	0	xxxx	01	STRHT
01	0	xxxx	11	Unallocated.
01	1	xxxx	01	LDRHT
01	1	xxxx	10	LDRSBT
01	1	xxxx	11	LDRSHT
10	0	!= 1111	10	LDRD (immediate) - Offset variant
10	0	xxxx	01	STRH (immediate) - Offset variant
10	0	xxxx	11	STRD (immediate) - Offset variant
10	1	!= 1111	01	LDRH (immediate) - Offset variant
10	1	!= 1111	10	LDRSB (immediate) - Offset variant
10	1	!= 1111	11	LDRSH (immediate) - Offset variant
11	0	!= 1111	10	LDRD (immediate) - Pre-indexed variant
11	0	xxxx	01	STRH (immediate) - Pre-indexed variant
11	0	xxxx	11	STRD (immediate) - Pre-indexed variant
11	1	!= 1111	01	LDRH (immediate) - Pre-indexed variant
11	1	!= 1111	10	LDRSB (immediate) - Pre-indexed variant
11	1	!= 1111	11	LDRSH (immediate) - Pre-indexed variant

C4.1.3 Synchronization primitives and Load-Acquire/Store-Release

This section describes the encoding of the Synchronization primitives and Load-Acquire/Store-Release group. The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

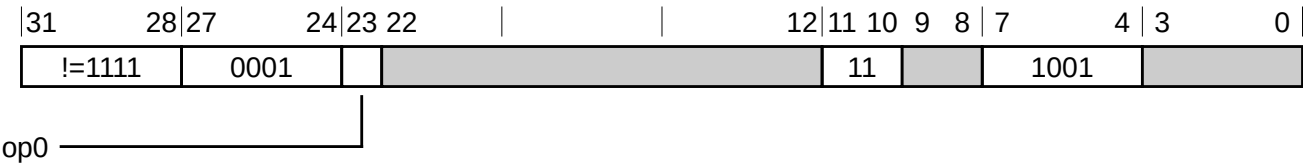


Table C4-8 Encoding table for the Synchronization primitives and Load-Acquire/Store-Release group

Decode fields	Decode group or instruction page
op0	
0	Unallocated.
1	Load/Store Exclusive and Load-Acquire/Store-Release

C4.1.3.1 Load/Store Exclusive and Load-Acquire/Store-Release

This section describes the encoding of the Load/Store Exclusive and Load-Acquire/Store-Release instruction class. The encodings in this section are decoded from [Synchronization primitives and Load-Acquire/Store-Release](#).

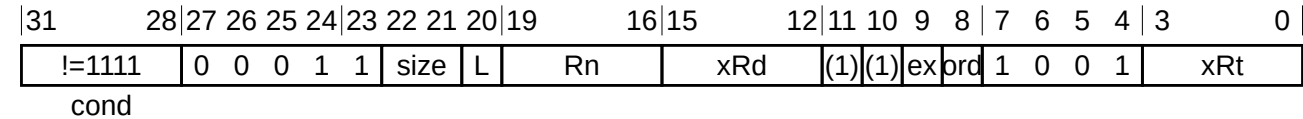


Table C4-9

Decode fields				Instruction page
size	L	ex	ord	
00	0	0	0	STL
00	0	0	1	Unallocated.
00	0	1	0	STLEX
00	0	1	1	STREX
00	1	0	0	LDA
00	1	0	1	Unallocated.
00	1	1	0	LDAEX
00	1	1	1	LDREX
01	0	0	x	Unallocated.
01	0	1	0	STLEXD
01	0	1	1	STREXD

Decode fields				Instruction page
size	L	ex	ord	
01	1	0	×	Unallocated.
01	1	1	0	LDAEXD
01	1	1	1	LDREXD
10	0	0	0	STLB
10	0	0	1	Unallocated.
10	0	1	0	STLEXB
10	0	1	1	STREXB
10	1	0	0	LDAB
10	1	0	1	Unallocated.
10	1	1	0	LDAEXB
10	1	1	1	LDREXB
11	0	0	0	STLH
11	0	0	1	Unallocated.
11	0	1	0	STLEXH
11	0	1	1	STREXH
11	1	0	0	LDAH
11	1	0	1	Unallocated.
11	1	1	0	LDAEXH
11	1	1	1	LDREXH

C4.1.4 Miscellaneous

This section describes the encoding of the Miscellaneous group. The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).

31	28	27	23	22	21	20	19			8	7	6	4	3	0
!=1111		00010	op0	0						0		op1			

Table C4-10 Encoding table for the Miscellaneous group

Decode fields		Decode group or instruction page
op0	op1	
00	001	Unallocated.
00	010	Unallocated.
00	011	Unallocated.

Decode fields		Decode group or instruction page
op0	op1	
00	110	Unallocated.
01	001	BX
01	010	BXJ
01	011	BLX (register)
01	110	Unallocated.
10	001	Unallocated.
10	010	Unallocated.
10	011	Unallocated.
10	110	Unallocated.
11	001	CLZ
11	010	Unallocated.
11	011	Unallocated.
11	110	ERET
xx	111	Exception Generation
xx	000	Move special register (register)
xx	100	Cyclic Redundancy Check
xx	101	Integer Saturating Arithmetic

C4.1.4.1 Exception Generation

This section describes the encoding of the Exception Generation instruction class. The encodings in this section are decoded from [Miscellaneous](#).

31	28	27	26	25	24	23	22	21	20	19				8	7	6	5	4	3	0		
!=1111				0	0	0	1	0	opc	0	imm12						0	1	1	1	imm4	
cond																						

Table C4-11

Decode fields	Instruction page
opc	
00	HLT
01	BKPT
10	HVC
11	SMC

C4.1.4.2 Move special register (register)

This section describes the encoding of the Move special register (register) instruction class. The encodings in this section are decoded from [Miscellaneous](#).

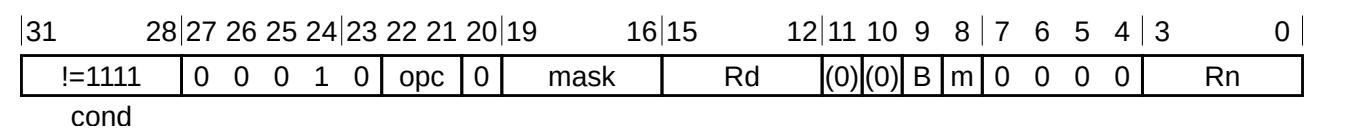


Table C4-12

Decode fields		Instruction page	
opc	B		
x0	0	MRS	
x0	1	MRS (Banked register)	
x1	0	MSR (register)	
x1	1	MSR (Banked register)	

C4.1.4.3 Cyclic Redundancy Check

This section describes the encoding of the Cyclic Redundancy Check instruction class. The encodings in this section are decoded from [Miscellaneous](#).

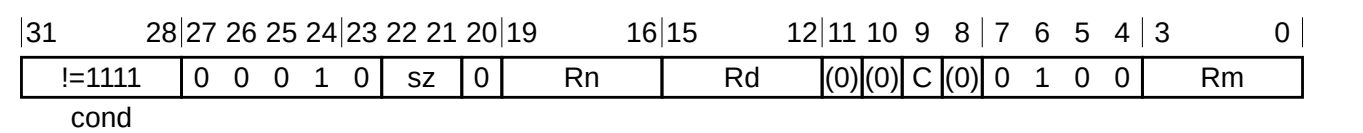


Table C4-13

Decode fields		Instruction page		Feature	
sz	C				
00	0	CRC32 - CRC32B variant		FEAT_CRC32	
00	1	CRC32C - CRC32CB variant		FEAT_CRC32	
01	0	CRC32 - CRC32H variant		FEAT_CRC32	
01	1	CRC32C - CRC32CH variant		FEAT_CRC32	
10	0	CRC32 - CRC32W variant		FEAT_CRC32	
10	1	CRC32C - CRC32CW variant		FEAT_CRC32	
11	x	CONSTRAINED UNPREDICTABLE		-	

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings](#).

C4.1.4.4 Integer Saturating Arithmetic

This section describes the encoding of the Integer Saturating Arithmetic instruction class. The encodings in this section are decoded from [Miscellaneous](#).

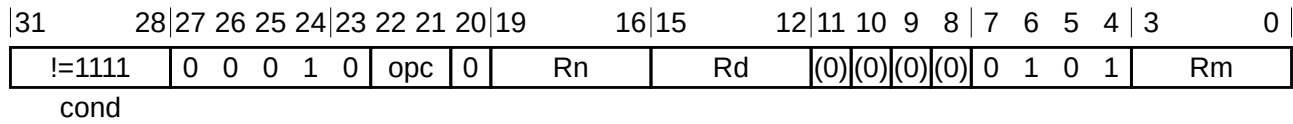
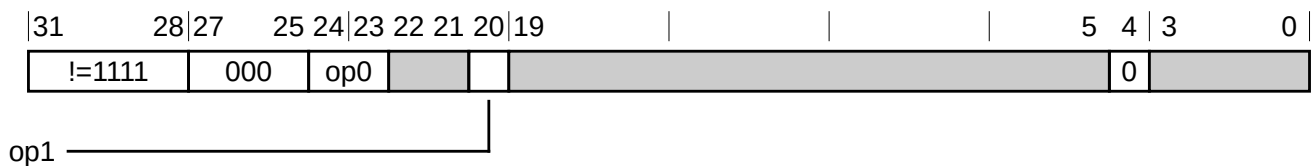


Table C4-14

Decode fields	Instruction page
opc	
00	QADD
01	QSUB
10	QDADD
11	QDSUB

C4.1.5 Data-processing register (immediate shift)

This section describes the encoding of the Data-processing register (immediate shift) group. The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).



This decode also imposes the constraint:

- $op0:op1 \neq 100$.

Table C4-15 Encoding table for the Data-processing register (immediate shift) group

Decode fields		Decode group or instruction page
op0	op1	
0x	x	Integer Data Processing (three register, immediate shift)
10	1	Integer Test and Compare (two register, immediate shift)

Decode fields		Decode group or instruction page
op0	op1	
11	x	Logical Arithmetic (three register, immediate shift)

C4.1.5.1 Integer Data Processing (three register, immediate shift)

This section describes the encoding of the Integer Data Processing (three register, immediate shift) instruction class. The encodings in this section are decoded from [Data-processing register \(immediate shift\)](#).

31	28	27	26	25	24	23	21	20	19	16	15	12	11	7	6	5	4	3	0
!=1111	0	0	0	0	opc	S	Rn		Rd	imm5	stype	0		Rm					
cond																			

Table C4-16

Decode fields				Instruction page
opc	S	Rn	imm5:stype	
000	x	xxxx	!= 0000011	AND, ANDS (register) - ANDS, shift or rotate by value variant
000	x	xxxx	0000011	AND, ANDS (register) - ANDS, rotate right with extend variant
001	x	xxxx	!= 0000011	EOR, EORS (register) - EORS, shift or rotate by value variant
001	x	xxxx	0000011	EOR, EORS (register) - EORS, rotate right with extend variant
010	0	!= 1101	!= 0000011	SUB, SUBS (register) - SUB, shift or rotate by value variant
010	0	!= 1101	0000011	SUB, SUBS (register) - SUB, rotate right with extend variant
010	0	1101	!= 0000011	SUB, SUBS (SP minus register) - SUB, shift or rotate by value variant
010	0	1101	0000011	SUB, SUBS (SP minus register) - SUB, rotate right with extend variant
010	1	!= 1101	!= 0000011	SUB, SUBS (register) - SUBS, shift or rotate by value variant
010	1	!= 1101	0000011	SUB, SUBS (register) - SUBS, rotate right with extend variant
010	1	1101	!= 0000011	SUB, SUBS (SP minus register) - SUBS, shift or rotate by value variant
010	1	1101	0000011	SUB, SUBS (SP minus register) - SUBS, rotate right with extend variant
011	x	xxxx	!= 0000011	RSB, RSBS (register) - RSBS, shift or rotate by value variant
011	x	xxxx	0000011	RSB, RSBS (register) - RSBS, rotate right with extend variant
100	0	!= 1101	!= 0000011	ADD, ADDS (register) - ADD, shift or rotate by value variant
100	0	!= 1101	0000011	ADD, ADDS (register) - ADD, rotate right with extend variant
100	0	1101	!= 0000011	ADD, ADDS (SP plus register) - ADD, shift or rotate by value variant
100	0	1101	0000011	ADD, ADDS (SP plus register) - ADD, rotate right with extend variant
100	1	!= 1101	!= 0000011	ADD, ADDS (register) - ADDS, shift or rotate by value variant
100	1	!= 1101	0000011	ADD, ADDS (register) - ADDS, rotate right with extend variant
100	1	1101	!= 0000011	ADD, ADDS (SP plus register) - ADDS, shift or rotate by value variant
100	1	1101	0000011	ADD, ADDS (SP plus register) - ADDS, rotate right with extend variant

Decode fields				Instruction page
opc	S	Rn	imm5:stype	
101	x	xxxx	!= 0000011	ADC, ADCS (register) - ADCS, shift or rotate by value variant
101	x	xxxx	0000011	ADC, ADCS (register) - ADCS, rotate right with extend variant
110	x	xxxx	!= 0000011	SBC, SBCS (register) - SBCS, shift or rotate by value variant
110	x	xxxx	0000011	SBC, SBCS (register) - SBCS, rotate right with extend variant
111	x	xxxx	!= 0000011	RSC, RSCS (register) - RSCS, shift or rotate by value variant
111	x	xxxx	0000011	RSC, RSCS (register) - RSCS, rotate right with extend variant

C4.1.5.2 Integer Test and Compare (two register, immediate shift)

This section describes the encoding of the Integer Test and Compare (two register, immediate shift) instruction class. The encodings in this section are decoded from [Data-processing register \(immediate shift\)](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	14	13	12	11	7	6	5	4	3	0	
!=1111				0 0 0 1 0		opc		1	Rn			(0)	(0)	(0)	(0)	imm5			stype		0	Rm	
cond																							

Table C4-17

Decode fields		Instruction page
opc	imm5:stype	
00	!= 0000011	TST (register) - Shift or rotate by value variant
00	0000011	TST (register) - Rotate right with extend variant
01	!= 0000011	TEQ (register) - Shift or rotate by value variant
01	0000011	TEQ (register) - Rotate right with extend variant
10	!= 0000011	CMP (register) - Shift or rotate by value variant
10	0000011	CMP (register) - Rotate right with extend variant
11	!= 0000011	CMN (register) - Shift or rotate by value variant
11	0000011	CMN (register) - Rotate right with extend variant

C4.1.5.3 Logical Arithmetic (three register, immediate shift)

This section describes the encoding of the Logical Arithmetic (three register, immediate shift) instruction class. The encodings in this section are decoded from [Data-processing register \(immediate shift\)](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	7	6	5	4	3	0
!=1111				0 0 0 1 1		opc		S	Rn			Rd		imm5		stype		0	Rm	
cond																				

Table C4-18

Decode fields		Instruction page
opc	imm5:stype	
00	!= 0000011	ORR, ORRS (register) - ORRS, shift or rotate by value variant
00	0000011	ORR, ORRS (register) - ORRS, rotate right with extend variant
01	!= 0000011	MOV, MOVS (register) - MOVS, shift or rotate by value variant
01	0000011	MOV, MOVS (register) - MOVS, rotate right with extend variant
10	!= 0000011	BIC, BICS (register) - BICS, shift or rotate by value variant
10	0000011	BIC, BICS (register) - BICS, rotate right with extend variant
11	!= 0000011	MVN, MVNS (register) - MVNS, shift or rotate by value variant
11	0000011	MVN, MVNS (register) - MVNS, rotate right with extend variant

C4.1.6 Data-processing register (register shift)

This section describes the encoding of the Data-processing register (register shift) group. The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).



This decode also imposes the constraint:

- op0:op1 != 100.

Table C4-19 Encoding table for the Data-processing register (register shift) group

Decode fields		Decode group or instruction page
op0	op1	
0x	x	Integer Data Processing (three register, register shift)
10	1	Integer Test and Compare (two register, register shift)
11	x	Logical Arithmetic (three register, register shift)

C4.1.6.1 Integer Data Processing (three register, register shift)

This section describes the encoding of the Integer Data Processing (three register, register shift) instruction class. The encodings in this section are decoded from [Data-processing register \(register shift\)](#).

31	28	27	26	25	24	23	21	20	19	16	15	12	11	8	7	6	5	4	3	0									
!=1111				0 0 0 0				opc		S		Rn			Rd			Rs			0		stype		1		Rm		
cond																													

Table C4-20

Decode fields	Instruction page
opc	
000	AND, ANDS (register-shifted register)
001	EOR, EORS (register-shifted register)
010	SUB, SUBS (register-shifted register)
011	RSB, RSBS (register-shifted register)
100	ADD, ADDS (register-shifted register)
101	ADC, ADCS (register-shifted register)
110	SBC, SBCS (register-shifted register)
111	RSC, RSCS (register-shifted register)

C4.1.6.2 Integer Test and Compare (two register, register shift)

This section describes the encoding of the Integer Test and Compare (two register, register shift) instruction class. The encodings in this section are decoded from [Data-processing register \(register shift\)](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	14	13	12	11	8	7	6	5	4	3	0
!=1111				0	0	0	1	0	opc	1	Rn		(0)	(0)	(0)	(0)	Rs		0	stype	1	Rm	
cond																							

Table C4-21

Decode fields	Instruction page
opc	
00	TST (register-shifted register)
01	TEQ (register-shifted register)
10	CMP (register-shifted register)
11	CMN (register-shifted register)

C4.1.6.3 Logical Arithmetic (three register, register shift)

This section describes the encoding of the Logical Arithmetic (three register, register shift) instruction class. The encodings in this section are decoded from [Data-processing register \(register shift\)](#).

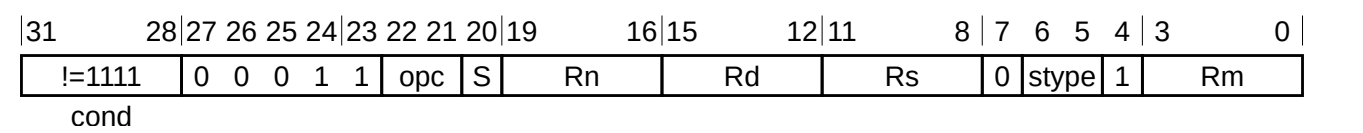


Table C4-22

Decode fields	Instruction page
opc	
00	ORR, ORRS (register-shifted register)
01	MOV, MOVS (register-shifted register)
10	BIC, BICS (register-shifted register)
11	MVN, MVNS (register-shifted register)

C4.1.7 Data-processing immediate

This section describes the encoding of the Data-processing immediate group. The encodings in this section are decoded from [Data-processing and miscellaneous instructions](#).



Table C4-23 Encoding table for the Data-processing immediate group

Decode fields		Decode group or instruction page
op0	op1	
0x	xx	Integer Data Processing (two register and immediate)
10	00	Move Halfword (immediate)
10	10	Move Special Register and Hints (immediate)
10	x1	Integer Test and Compare (one register and immediate)
11	xx	Logical Arithmetic (two register and immediate)

C4.1.7.1 Integer Data Processing (two register and immediate)

This section describes the encoding of the Integer Data Processing (two register and immediate) instruction class. The encodings in this section are decoded from [Data-processing immediate](#).

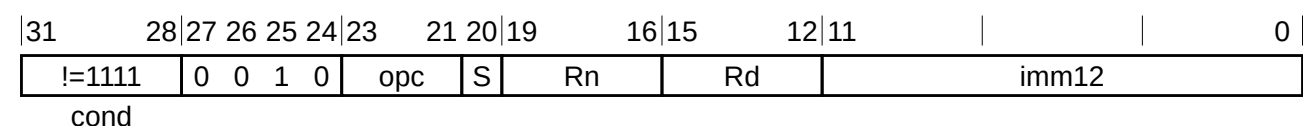


Table C4-24

Decode fields			Instruction page
opc	S	Rn	
000	x	xxxx	AND, ANDS (immediate)
001	x	xxxx	EOR, EORS (immediate)
010	0	!= 11x1	SUB, SUBS (immediate) - SUB variant
010	0	1101	SUB, SUBS (SP minus immediate) - SUB variant
010	0	1111	ADR - A2
010	1	!= 1101	SUB, SUBS (immediate) - SUBS variant
010	1	1101	SUB, SUBS (SP minus immediate) - SUBS variant
011	x	xxxx	RSB, RSBS (immediate)
100	0	!= 11x1	ADD, ADDS (immediate) - ADD variant
100	0	1101	ADD, ADDS (SP plus immediate) - ADD variant
100	0	1111	ADR - A1
100	1	!= 1101	ADD, ADDS (immediate) - ADDS variant
100	1	1101	ADD, ADDS (SP plus immediate) - ADDS variant
101	x	xxxx	ADC, ADCS (immediate)
110	x	xxxx	SBC, SBCS (immediate)
111	x	xxxx	RSC, RSCS (immediate)

C4.1.7.2 Move Halfword (immediate)

This section describes the encoding of the Move Halfword (immediate) instruction class. The encodings in this section are decoded from [Data-processing immediate](#).

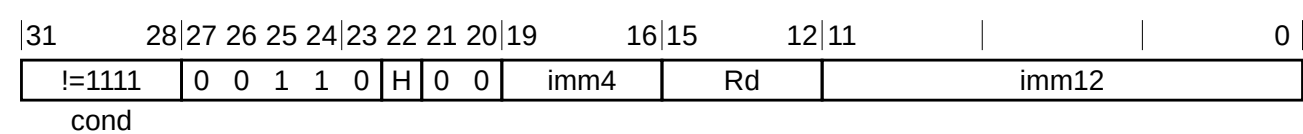


Table C4-25

Decode fields	Instruction page
H	
0	MOV, MOVS (immediate)
1	MOVT

C4.1.7.3 Move Special Register and Hints (immediate)

This section describes the encoding of the Move Special Register and Hints (immediate) instruction class. The encodings in this section are decoded from [Data-processing immediate](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	14	13	12	11				0	
!=1111				0 0 1 1 0				R	1	0	imm4			(1)	(1)	(1)	(1)	imm12			
cond																					

Table C4-26

Decode fields		Instruction page
R:imm4	imm12	
!= 00000	xxxxxxxxxxxxx	MSR (immediate)
00000	xxxx00000000	NOP
00000	xxxx00000001	YIELD
00000	xxxx00000010	WFE
00000	xxxx00000011	WFI
00000	xxxx00000100	SEV
00000	xxxx00000101	SEVL
00000	xxxx0000011x	Reserved hint, behaves as NOP.
00000	xxxx00001xxx	Reserved hint, behaves as NOP.
00000	xxxx00010000	Reserved hint, behaves as NOP.
00000	xxxx000100xx	Reserved hint, behaves as NOP.
00000	xxxx00010101	Reserved hint, behaves as NOP.
00000	xxxx00010111	Reserved hint, behaves as NOP.
00000	xxxx00011xxx	Reserved hint, behaves as NOP.
00000	xxxx001xxxxx	Reserved hint, behaves as NOP.
00000	xxxx01xxxxxx	Reserved hint, behaves as NOP.
00000	xxxx10xxxxxx	Reserved hint, behaves as NOP.
00000	xxxx110xxxxx	Reserved hint, behaves as NOP.
00000	xxxx1110xxxx	Reserved hint, behaves as NOP.
00000	xxxx1111xxxx	DBG

C4.1.7.4 Integer Test and Compare (one register and immediate)

This section describes the encoding of the Integer Test and Compare (one register and immediate) instruction class. The encodings in this section are decoded from [Data-processing immediate](#).

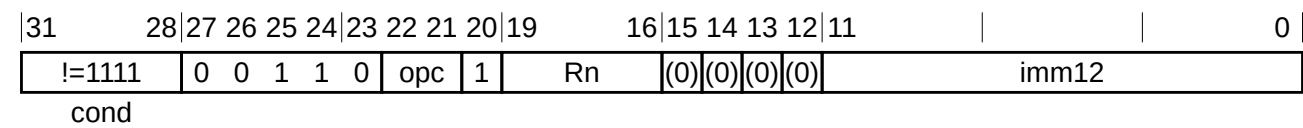


Table C4-27

Decode fields	Instruction page
opc	
00	TST (immediate)
01	TEQ (immediate)
10	CMP (immediate)
11	CMN (immediate)

C4.1.7.5 Logical Arithmetic (two register and immediate)

This section describes the encoding of the Logical Arithmetic (two register and immediate) instruction class. The encodings in this section are decoded from [Data-processing immediate](#).

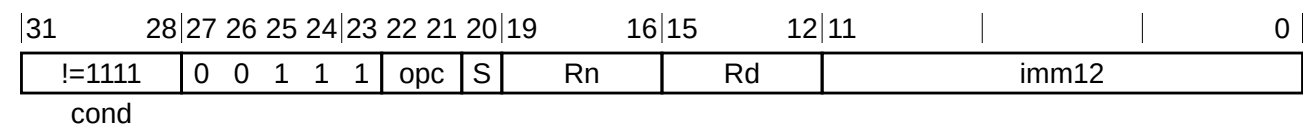


Table C4-28

Decode fields	Instruction page
opc	
00	ORR, ORRS (immediate)
01	MOV, MOVS (immediate)
10	BIC, BICS (immediate)
11	MVN, MVNS (immediate)

C4.1.8 Load/Store Word, Unsigned Byte (immediate, literal)

This section describes the encoding of the Load/Store Word, Unsigned Byte (immediate, literal) instruction class. The encodings in this section are decoded from [A32 instruction set encoding](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11			0
!=1111	0	1	0	P	U	o2	W	o1		Rn		Rt					imm12
cond																	

Table C4-29

Decode fields				Instruction page
P:W	o2	o1	Rn	
!= 01	0	1	1111	LDR (literal)
!= 01	1	1	1111	LDRB (literal)
00	0	0	xxxx	STR (immediate) - Post-indexed variant
00	0	1	!= 1111	LDR (immediate) - Post-indexed variant
00	1	0	xxxx	STRB (immediate) - Post-indexed variant
00	1	1	!= 1111	LDRB (immediate) - Post-indexed variant
01	0	0	xxxx	STRT
01	0	1	xxxx	LDRT
01	1	0	xxxx	STRBT
01	1	1	xxxx	LDRBT
10	0	0	xxxx	STR (immediate) - Offset variant
10	0	1	!= 1111	LDR (immediate) - Offset variant
10	1	0	xxxx	STRB (immediate) - Offset variant
10	1	1	!= 1111	LDRB (immediate) - Offset variant
11	0	0	xxxx	STR (immediate) - Pre-indexed variant
11	0	1	!= 1111	LDR (immediate) - Pre-indexed variant
11	1	0	xxxx	STRB (immediate) - Pre-indexed variant
11	1	1	!= 1111	LDRB (immediate) - Pre-indexed variant

C4.1.9 Load/Store Word, Unsigned Byte (register)

This section describes the encoding of the Load/Store Word, Unsigned Byte (register) instruction class. The encodings in this section are decoded from [A32 instruction set encoding](#).

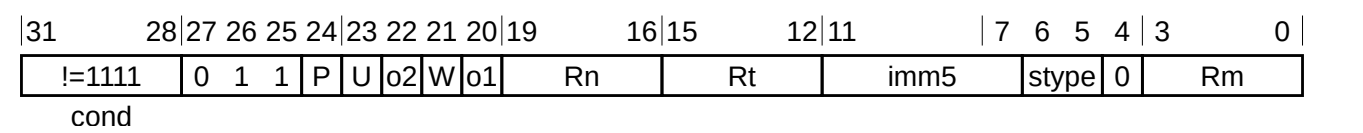


Table C4-30

Decode fields				Instruction page
P	o2	W	o1	
0	0	0	0	STR (register) - Post-indexed variant
0	0	0	1	LDR (register) - Post-indexed variant
0	0	1	0	STRT
0	0	1	1	LDRT
0	1	0	0	STRB (register) - Post-indexed variant
0	1	0	1	LDRB (register) - Post-indexed variant
0	1	1	0	STRBT
0	1	1	1	LDRBT
1	0	x	0	STR (register) - Pre-indexed variant
1	0	x	1	LDR (register) - Pre-indexed variant
1	1	x	0	STRB (register) - Pre-indexed variant
1	1	x	1	LDRB (register) - Pre-indexed variant

C4.1.10 Media instructions

This section describes the encoding of the Media instructions group. The encodings in this section are decoded from [A32 instruction set encoding](#).



Table C4-31 Encoding table for the Media instructions group

Decode fields		Decode group or instruction page
op0	op1	
00xxx	xxx	Parallel Arithmetic
01000	101	SEL
01000	001	Unallocated.
01000	xx0	PKHBT, PKHTB
01001	x01	Unallocated.
01001	xx0	Unallocated.
0110x	x01	Unallocated.
0110x	xx0	Unallocated.
01x10	001	Saturate 16-bit
01x10	101	Unallocated.
01x11	x01	Reverse Bit/Byte
01x1x	xx0	Saturate 32-bit
01xxx	111	Unallocated.
01xxx	011	Extend and Add
10xxx	xxx	Signed multiply, Divide
11000	000	Unsigned Sum of Absolute Differences
11000	100	Unallocated.
11001	x00	Unallocated.
1101x	x00	Unallocated.
110xx	111	Unallocated.
1110x	111	Unallocated.
1110x	x00	Bitfield Insert
11110	111	Unallocated.
11111	111	Permanently UNDEFINED
1111x	x00	Unallocated.
11x0x	x10	Unallocated.
11x1x	x10	Bitfield Extract
11xxx	011	Unallocated.
11xxx	x01	Unallocated.

C4.1.10.1 Parallel Arithmetic

This section describes the encoding of the Parallel Arithmetic instruction class. The encodings in this section are decoded from [Media instructions](#).

31	28	27	26	25	24	23	22	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0	
!=1111				0 1 1 0 0				op1		Rn		Rd		(1)(1)(1)(1)		B	op2		1		Rm		
cond																							

Table C4-32

Decode fields			Instruction page
op1	B	op2	
000	x	xx	Unallocated.
001	0	00	SADD16
001	0	01	SASX
001	0	10	SSAX
001	0	11	SSUB16
001	1	00	SADD8
001	1	01	Unallocated.
001	1	10	Unallocated.
001	1	11	SSUB8
010	0	00	QADD16
010	0	01	QASX
010	0	10	QSAX
010	0	11	QSUB16
010	1	00	QADD8
010	1	01	Unallocated.
010	1	10	Unallocated.
010	1	11	QSUB8
011	0	00	SHADD16
011	0	01	SHASX
011	0	10	SHSAX
011	0	11	SHSUB16
011	1	00	SHADD8
011	1	01	Unallocated.
011	1	10	Unallocated.
011	1	11	SHSUB8
100	x	xx	Unallocated.
101	0	00	UADD16
101	0	01	UASX
101	0	10	USAX
101	0	11	USUB16
101	1	00	UADD8
101	1	01	Unallocated.
101	1	10	Unallocated.
101	1	11	USUB8
110	0	00	UQADD16
110	0	01	UQASX

Decode fields			Instruction page
op1	B	op2	
110	0	10	UQSAX
110	0	11	UQSUB16
110	1	00	UQADD8
110	1	01	Unallocated.
110	1	10	Unallocated.
110	1	11	UQSUB8
111	0	00	UHADD16
111	0	01	UHASX
111	0	10	UHSAX
111	0	11	UHSUB16
111	1	00	UHADD8
111	1	01	Unallocated.
111	1	10	Unallocated.
111	1	11	UHSUB8

C4.1.10.2 Saturate 16-bit

This section describes the encoding of the Saturate 16-bit instruction class. The encodings in this section are decoded from [Media instructions](#).

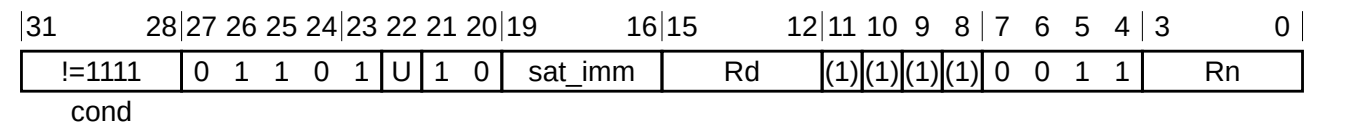


Table C4-33

Decode fields	Instruction page
U	
0	SSAT16
1	USAT16

C4.1.10.3 Reverse Bit/Byte

This section describes the encoding of the Reverse Bit/Byte instruction class. The encodings in this section are decoded from [Media instructions](#).

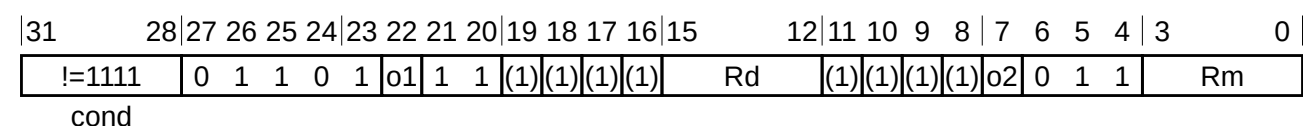


Table C4-34

Decode fields		Instruction page
o1	o2	
0	0	REV
0	1	REV16
1	0	RBIT
1	1	REVSH

C4.1.10.4 Saturate 32-bit

This section describes the encoding of the Saturate 32-bit instruction class. The encodings in this section are decoded from [Media instructions](#).

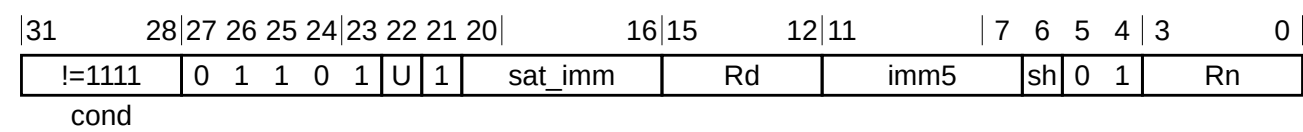


Table C4-35

Decode fields		Instruction page
U		
0		SSAT
1		USAT

C4.1.10.5 Extend and Add

This section describes the encoding of the Extend and Add instruction class. The encodings in this section are decoded from [Media instructions](#).

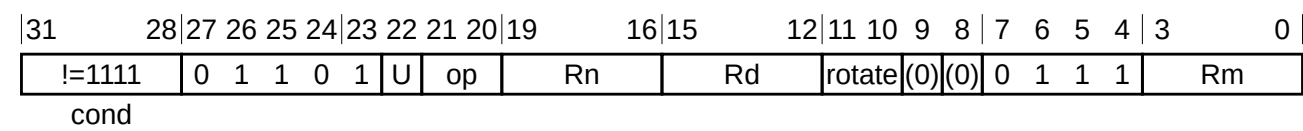


Table C4-36

Decode fields			Instruction page
U	op	Rn	
0	00	!= 1111	SXTAB16
0	00	1111	SXTB16
0	10	!= 1111	SXTAB
0	10	1111	SXTB
0	11	!= 1111	SXTAH
0	11	1111	SXTH
1	00	!= 1111	UXTAB16
1	00	1111	UXTB16
1	10	!= 1111	UXTAB
1	10	1111	UXTB
1	11	!= 1111	UXTAH
1	11	1111	UXTH

C4.1.10.6 Signed multiply, Divide

This section describes the encoding of the Signed multiply, Divide instruction class. The encodings in this section are decoded from [Media instructions](#).

31	28	27	26	25	24	23	22	20	19	16	15	12	11	8	7	5	4	3	0	
!=1111				0	1	1	1	0	op1	Rd		Ra		Rm		op2		1	Rn	
cond																				

Table C4-37

Decode fields			Instruction page
op1	Ra	op2	
000	!= 1111	000	SMLAD, SMLADX - SMLAD variant
000	!= 1111	001	SMLAD, SMLADX - SMLADX variant
000	!= 1111	010	SMLSD, SMLSDX - SMLSD variant
000	!= 1111	011	SMLSD, SMLSDX - SMLSDX variant
000	xxxx	1xx	Unallocated.
000	1111	000	SMUAD, SMUADX - SMUAD variant
000	1111	001	SMUAD, SMUADX - SMUADX variant
000	1111	010	SMUSD, SMUSDX - SMUSD variant

Decode fields			Instruction page
op1	Ra	op2	
000	1111	011	SMUSD, SMUSDx - SMUSDx variant
001	xxxx	000	SDIV
001	xxxx	!= 000	Unallocated.
010	xxxx	xxx	Unallocated.
011	xxxx	000	UDIV
011	xxxx	!= 000	Unallocated.
100	xxxx	000	SMLALD, SMLALDX - SMLALD variant
100	xxxx	001	SMLALD, SMLALDX - SMLALDX variant
100	xxxx	010	SMLSLD, SMLSLDX - SMLSLD variant
100	xxxx	011	SMLSLD, SMLSLDX - SMLSLDX variant
100	xxxx	1xx	Unallocated.
101	!= 1111	000	SMMLA, SMMLAR - SMMLA variant
101	!= 1111	001	SMMLA, SMMLAR - SMMLAR variant
101	xxxx	01x	Unallocated.
101	xxxx	10x	Unallocated.
101	xxxx	110	SMMLS, SMMLSR - SMMLS variant
101	xxxx	111	SMMLS, SMMLSR - SMMLSR variant
101	1111	000	SMMUL, SMMULR - SMMUL variant
101	1111	001	SMMUL, SMMULR - SMMULR variant
11x	xxxx	xxx	Unallocated.

C4.1.10.7 Unsigned Sum of Absolute Differences

This section describes the encoding of the Unsigned Sum of Absolute Differences instruction class. The encodings in this section are decoded from [Media instructions](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
!=1111				0	1	1	1	1	0	0	0	Rd	Ra	Rm	0	0	0	1	Rn		
cond																					

Table C4-38

Decode fields	Instruction page
Ra	
!= 1111	USADA8
1111	USAD8

C4.1.10.8 Bitfield Insert

This section describes the encoding of the Bitfield Insert instruction class. The encodings in this section are decoded from [Media instructions](#).

31	28	27	26	25	24	23	22	21	20	16	15	12	11	7	6	5	4	3	0						
!=1111				0	1	1	1	1	1	0	msb				Rd		lsb			0	0	1	Rn		
cond																									

Table C4-39

Decode fields	Instruction page
Rn	
!= 1111	BFI
1111	BFC

C4.1.10.9 Permanently UNDEFINED

This section describes the encoding of the Permanently UNDEFINED instruction class. The encodings in this section are decoded from [Media instructions](#).

31	28	27	26	25	24	23	22	21	20	19				8	7	6	5	4	3		0																														
!=1111				0				1				1				1				1				1				imm12				1				1				1				1				imm4			
cond																																																			

Table C4-40

Decode fields	Instruction page
cond	
0xxx	Unallocated.
10xx	Unallocated.
110x	Unallocated.
1110	UDF

C4.1.10.10 Bitfield Extract

This section describes the encoding of the Bitfield Extract instruction class. The encodings in this section are decoded from [Media instructions](#).

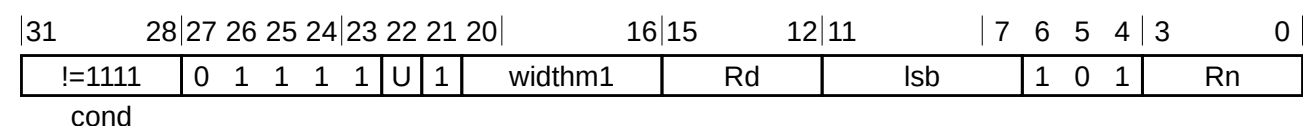


Table C4-41

Decode fields	Instruction page
U	
0	SBFX
1	UBFX

C4.1.11 Branch, branch with link, and block data transfer

This section describes the encoding of the Branch, branch with link, and block data transfer group. The encodings in this section are decoded from [A32 instruction set encoding](#).

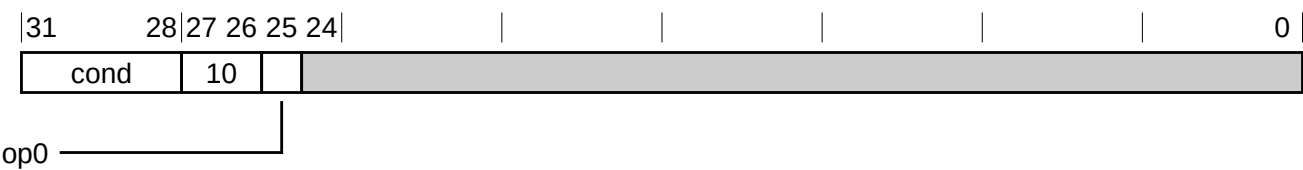


Table C4-42 Encoding table for the Branch, branch with link, and block data transfer group

Decode fields	Decode group or instruction page
condop0	
11110	Exception Save/Restore
!= 11110	Load/Store Multiple
xxxx1	Branch (immediate)

C4.1.11.1 Exception Save/Restore

This section describes the encoding of the Exception Save/Restore instruction class. The encodings in this section are decoded from [Branch, branch with link, and block data transfer](#).

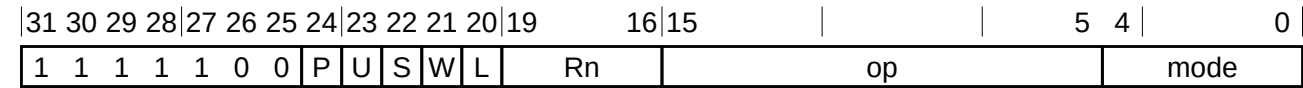


Table C4-43

Decode fields				Instruction page
P	U	S	L	
x	x	0	0	Unallocated.
0	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Decrement After variant
0	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Decrement After variant
0	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Increment After variant
0	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Increment After variant
1	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Decrement Before variant
1	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Decrement Before variant
x	x	1	1	Unallocated.
1	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB - Increment Before variant
1	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB - Increment Before variant

C4.1.11.2 Load/Store Multiple

This section describes the encoding of the Load/Store Multiple instruction class. The encodings in this section are decoded from Branch, branch with link, and block data transfer.

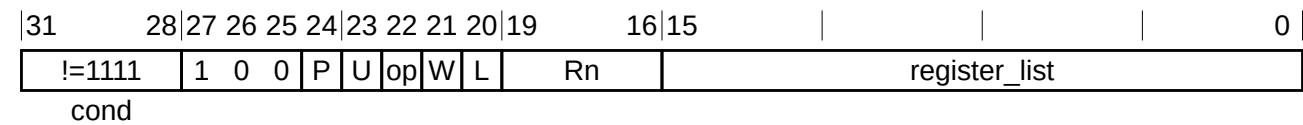


Table C4-44

Decode fields				Instruction page
P	U	op	L	register_list
0	0	0	0	STMDA, STMED
0	0	0	1	LDMDA, LDMFA
0	1	0	0	STM, STMIA, STMEA
0	1	0	1	LDM, LDMIA, LDMFD

Decode fields					Instruction page
P	U	op	L	register_list	
x	x	1	0	xxxxxxxxxxxxxxxx	STM (User registers)
1	0	0	0	xxxxxxxxxxxxxxxx	STMDB, STMFD
1	0	0	1	xxxxxxxxxxxxxxxx	LDMDB, LDMEA
x	x	1	1	0xxxxxxxxxxxxxxxx	LDM (User registers)
1	1	0	0	xxxxxxxxxxxxxxxx	STMIB, STMFA
1	1	0	1	xxxxxxxxxxxxxxxx	LDMIB, LDMED
x	x	1	1	1xxxxxxxxxxxxxxxx	LDM (exception return)

C4.1.11.3 Branch (immediate)

This section describes the encoding of the Branch (immediate) instruction class. The encodings in this section are decoded from [Branch](#), [branch with link](#), and [block data transfer](#).

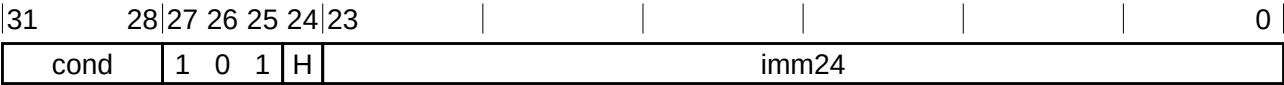


Table C4-45

Decode fields		Instruction page
cond	H	
!= 1111	0	B
!= 1111	1	BL, BLX (immediate) - A1
1111	x	BL, BLX (immediate) - A2

C4.1.12 System register access, Advanced SIMD, floating-point, and Supervisor call

This section describes the encoding of the System register access, Advanced SIMD, floating-point, and Supervisor call group. The encodings in this section are decoded from [A32 instruction set encoding](#).



Table C4-46 Encoding table for the System register access, Advanced SIMD, floating-point, and Supervisor call group

Decode fields				Decode group or instruction page
cond	op0	op1	op2	
xxxx	0x	0x	x	Unallocated.
xxxx	10	0x	x	Unallocated.
xxxx	11	xx	x	Supervisor call
1111	!= 11	1x	x	Unconditional Advanced SIMD and floating-point instructions
!= 1111	0x	1x	x	Advanced SIMD and System register load/store and 64-bit move
!= 1111	10	1x	1	Advanced SIMD and System register 32-bit move
!= 1111	10	10	0	Floating-point data-processing
!= 1111	10	11	0	Unallocated.

C4.1.13 Supervisor call

This section describes the encoding of the Supervisor call group. The encodings in this section are decoded from [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

31	28	27	24	23						0
cond	1111									

Table C4-47 Encoding table for the Supervisor call group

Decode fields	Decode group or instruction page
cond	
1111	Unallocated.
!= 1111	SVC

C4.1.14 Unconditional Advanced SIMD and floating-point instructions

This section describes the encoding of the Unconditional Advanced SIMD and floating-point instructions group. The encodings in this section are decoded from [System register access, Advanced SIMD, floating-point, and Supervisor call](#).



This decode also imposes the constraint:

- $\text{op0} \langle 2:1 \rangle \neq 11$.

Table C4-48 Encoding table for the Unconditional Advanced SIMD and floating-point instructions group

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0xx	xxxxxxx	x	xx	x	x	Unallocated.
100	xxxxxxx	0	xx	1	0	Unallocated.
100	xxxxxxx	0	!= 00	0	0	Floating-point conditional select
101	00xxxxx	0	!= 00	x	0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Unallocated.
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
101	11xxxx	0	!=00	0	0	Unallocated.
101	xxxxxxx	0	00	1	0	Unallocated.
10x	xxxxxxx	0	00	0	x	Unallocated.
10x	xxxxxxx	x	xx	x	1	Unallocated.

C4.1.14.1 Floating-point conditional select

This section describes the encoding of the Floating-point conditional select instruction class. The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

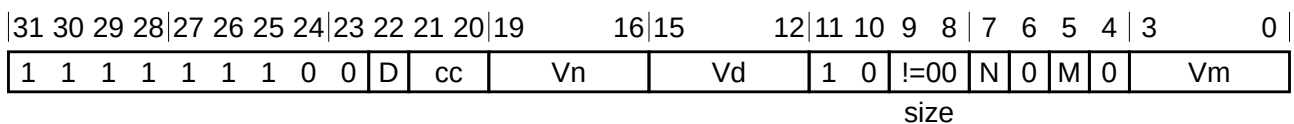


Table C4-49

Decode fields size	Instruction page
01	Unallocated.
10	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, single-precision scalar variant
11	VSELEQ, VSELGE, VSELGT, VSELVS - Greater than, double-precision scalar variant

C4.1.14.2 Floating-point minNum/maxNum

This section describes the encoding of the Floating-point minNum/maxNum instruction class. The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	!=00	N	op	M	0	Vm				

size

Table C4-50

Decode fields size	op	Instruction page
01	x	Unallocated.
10	0	VMAXNM - Single-precision scalar variant
10	1	VMINNM - Single-precision scalar variant
11	0	VMAXNM - Double-precision scalar variant
11	1	VMINNM - Double-precision scalar variant

C4.1.14.3 Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from [Unconditional Advanced SIMD and floating-point instructions](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	1	1	1	0	1	D	1	1	1	01	RM	Vd	1	0	!=00	op	1	M	0	Vm				

size

Table C4-51

Decode fields				Instruction page
o1	RM	size	op	
0	xx	1x	1	Unallocated.
0	xx	01	x	Unallocated.
0	00	10	0	VRINTA (floating-point) - Single-precision scalar variant
0	00	11	0	VRINTA (floating-point) - Double-precision scalar variant
0	01	10	0	VRINTN (floating-point) - Single-precision scalar variant
0	01	11	0	VRINTN (floating-point) - Double-precision scalar variant
0	10	10	0	VRINTP (floating-point) - Single-precision scalar variant
0	10	11	0	VRINTP (floating-point) - Double-precision scalar variant
0	11	10	0	VRINTM (floating-point) - Single-precision scalar variant
0	11	11	0	VRINTM (floating-point) - Double-precision scalar variant
1	00	10	x	VCVTA (floating-point) - Single-precision scalar variant
1	00	11	x	VCVTA (floating-point) - Double-precision scalar variant
1	01	10	x	VCVTN (floating-point) - Single-precision scalar variant
1	01	11	x	VCVTN (floating-point) - Double-precision scalar variant
1	10	10	x	VCVTP (floating-point) - Single-precision scalar variant
1	10	11	x	VCVTP (floating-point) - Double-precision scalar variant
1	11	10	x	VCVTM (floating-point) - Single-precision scalar variant
1	11	11	x	VCVTM (floating-point) - Double-precision scalar variant

C4.1.15 Advanced SIMD and System register load/store and 64-bit move

This section describes the encoding of the Advanced SIMD and System register load/store and 64-bit move group. The encodings in this section are decoded from [System register access, Advanced SIMD, floating-point, and Supervisor call](#).

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see [About the A32 Advanced SIMD and floating-point instructions and their encoding](#)

31	28	27	25	24	21	20		12	11	10	9	8		0
!=1111	110	op0						1	op1					

Table C4-52 Encoding table for the Advanced SIMD and System register load/store and 64-bit move group

Decode fields		Decode group or instruction page
op0	op1	
00x0	0x	Advanced SIMD and floating-point 64-bit move
00x0	11	System register 64-bit move
!= 00x0	0x	Advanced SIMD and floating-point load/store
!= 00x0	11	System register load/store
xxxx	10	Unallocated.

C4.1.15.1 Advanced SIMD and floating-point 64-bit move

This section describes the encoding of the Advanced SIMD and floating-point 64-bit move instruction class. The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
!=1111	1	1	0	0	0	D	0	op		Rt2		Rt		1	0	size	opc2	M	o3				Vm
cond																							

Table C4-53

Decode fields					Instruction page
D	op	size	opc2	o3	
0	x	xx	xx	x	Unallocated.
1	x	xx	xx	0	Unallocated.
1	x	0x	00	1	Unallocated.
1	x	xx	01	x	Unallocated.
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - From general-purpose registers variant
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - From general-purpose registers variant
1	x	xx	1x	x	Unallocated.
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) - To general-purpose registers variant
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) - To general-purpose registers variant

C4.1.15.2 System register 64-bit move

This section describes the encoding of the System register 64-bit move instruction class. The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

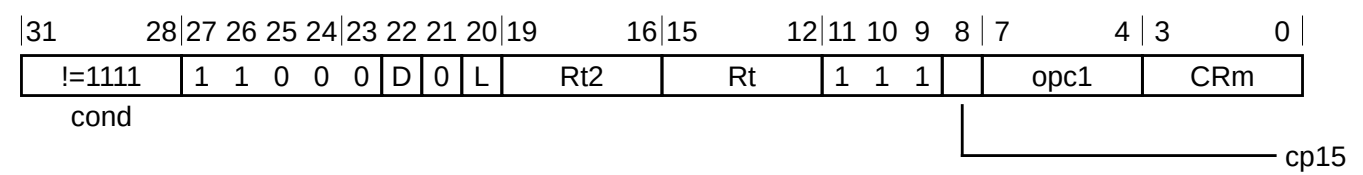


Table C4-54

Decode fields		Instruction page	
D	L		
0	x	Unallocated.	
1	0	MCRR	
1	1	MRRC	

C4.1.15.3 Advanced SIMD and floating-point load/store

This section describes the encoding of the Advanced SIMD and floating-point load/store instruction class. The encodings in this section are decoded from Advanced SIMD and System register load/store and 64-bit move.

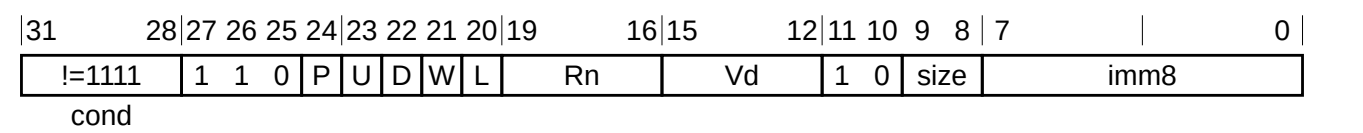


Table C4-55

Decode fields							Instruction page	
P	U	W	L	Rn	size	imm8		
0	0	1	x	xxxx	xx	xxxxxxxx	Unallocated.	
x	1	x	x	xxxx	0x	xxxxxxxx	Unallocated.	
0	1	x	0	xxxx	10	xxxxxxxx	VSTM, VSTMDB, VSTMIA - Increment After variant	
0	1	x	0	xxxx	11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA - Increment After variant	
0	1	x	0	xxxx	11	xxxxxxxx1	FSTMDBX, FSTMIAX - Increment After variant	
0	1	x	1	xxxx	10	xxxxxxxx	VLDM, VLDMDB, VLDMIA - Increment After variant	
0	1	x	1	xxxx	11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA - Increment After variant	
0	1	x	1	xxxx	11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - Increment After variant	
1	x	0	0	xxxx	10	xxxxxxxx	VSTR - Single-precision scalar variant	

Decode fields						Instruction page	
P	U	W	L	Rn	size	imm8	
1	x	0	0	xxxx	11	xxxxxxxx	VSTR - Double-precision scalar variant
1	x	0	1	!= 1111	10	xxxxxxxx	VLDR (immediate) - Single-precision scalar variant
1	x	0	1	!= 1111	11	xxxxxxxx	VLDR (immediate) - Double-precision scalar variant
1	0	1	x	xxxx	0x	xxxxxxxx	Unallocated.
1	0	1	0	xxxx	10	xxxxxxxx	VSTM, VSTMDB, VSTMIA - Decrement Before variant
1	0	1	0	xxxx	11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA - Decrement Before variant
1	0	1	0	xxxx	11	xxxxxxxx1	FSTMDBX, FSTMIA - Decrement Before variant
1	0	1	1	xxxx	10	xxxxxxxx	VLDM, VLDMDB, VLDMIA - Decrement Before variant
1	0	1	1	xxxx	11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA - Decrement Before variant
1	0	1	1	xxxx	11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) - Decrement Before variant
1	x	0	1	1111	10	xxxxxxxx	VLDR (literal) - Single-precision scalar variant
1	x	0	1	1111	11	xxxxxxxx	VLDR (literal) - Double-precision scalar variant
1	1	1	x	xxxx	xx	xxxxxxxx	Unallocated.

C4.1.15.4 System register load/store

This section describes the encoding of the System register load/store instruction class. The encodings in this section are decoded from [Advanced SIMD and System register load/store and 64-bit move](#).

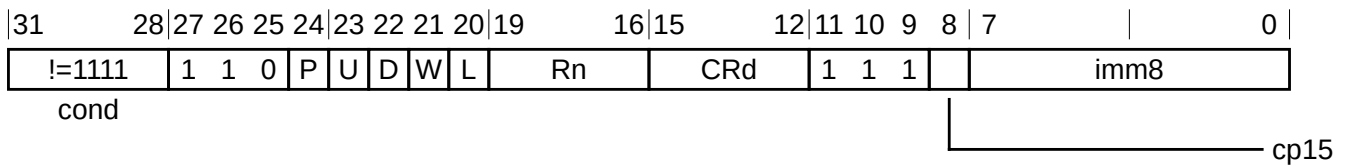


Table C4-56

Decode fields						Instruction page	
P:U:W	D	L	Rn	CRd	cp15		
!= 000	0	x	xxxx	!= 0101	0	Unallocated.	
!= 000	0	1	1111	0101	0	LDC (literal)	
!= 000	x	x	xxxx	xxxx	1	Unallocated.	
!= 000	1	x	xxxx	0101	0	Unallocated.	
0x1	0	0	xxxx	0101	0	STC - Post-indexed variant	
0x1	0	1	!= 1111	0101	0	LDC (immediate) - Post-indexed variant	

Decode fields						Instruction page
P:U:W	D	L	Rn	CRd	cp15	
010	0	0	xxxx	0101	0	STC - Unindexed variant
010	0	1	!= 1111	0101	0	LDC (immediate) - Unindexed variant
1x0	0	0	xxxx	0101	0	STC - Offset variant
1x0	0	1	!= 1111	0101	0	LDC (immediate) - Offset variant
1x1	0	0	xxxx	0101	0	STC - Pre-indexed variant
1x1	0	1	!= 1111	0101	0	LDC (immediate) - Pre-indexed variant

C4.1.16 Advanced SIMD and System register 32-bit move

This section describes the encoding of the Advanced SIMD and System register 32-bit move group. The encodings in this section are decoded from [System register access](#), [Advanced SIMD](#), [floating-point](#), and [Supervisor call](#).

31	28	27	24	23	21	20				12	11	10	8	7	5	4	3	0
!=1111		1110		op0						1		op1				1		

Table C4-57 Encoding table for the Advanced SIMD and System register 32-bit move group

Decode fields		Decode group or instruction page
op0	op1	
000	00x	Unallocated.
000	010	VMOV (between general-purpose register and single-precision)
001	010	Unallocated.
01x	010	Unallocated.
10x	010	Unallocated.
110	010	Unallocated.
111	010	Floating-point move special register
xxx	011	Advanced SIMD 8/16/32-bit element move/duplicate
xxx	10x	Unallocated.
xxx	11x	System register 32-bit move

C4.1.16.1 Floating-point move special register

This section describes the encoding of the Floating-point move special register instruction class. The encodings in this section are decoded from [Advanced SIMD](#) and [System register 32-bit move](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	2	1	0				
!=1111				1	1	1	0	1	1	1	L	reg			Rt			1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																													

Table C4-58

Decode fields																Instruction page															
L																															
0																VMSR															
1																VMRS															

C4.1.16.2 Advanced SIMD 8/16/32-bit element move/duplicate

This section describes the encoding of the Advanced SIMD 8/16/32-bit element move/duplicate instruction class. The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

31	28	27	26	25	24	23	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	2	1	0				
!=1111				1	1	1	0	opc1		L	Vn			Rt			1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)
cond																												

Table C4-59

Decode fields			Instruction page
opc1	L	opc2	
0xx	0	xx	VMOV (general-purpose register to scalar)
xxx	1	xx	VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	Unallocated.

C4.1.16.3 System register 32-bit move

This section describes the encoding of the System register 32-bit move instruction class. The encodings in this section are decoded from [Advanced SIMD and System register 32-bit move](#).

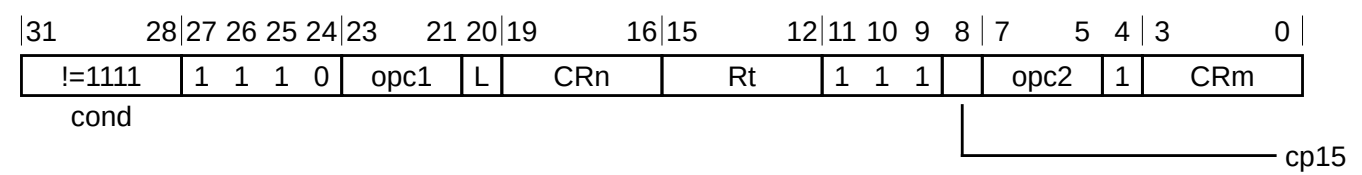


Table C4-60

Decode fields	Instruction page
L	
0	MCR
1	MRC

C4.1.17 Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from System register access, Advanced SIMD, floating-point, and Supervisor call.
This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see About the A32 Advanced SIMD and floating-point instructions and their encoding

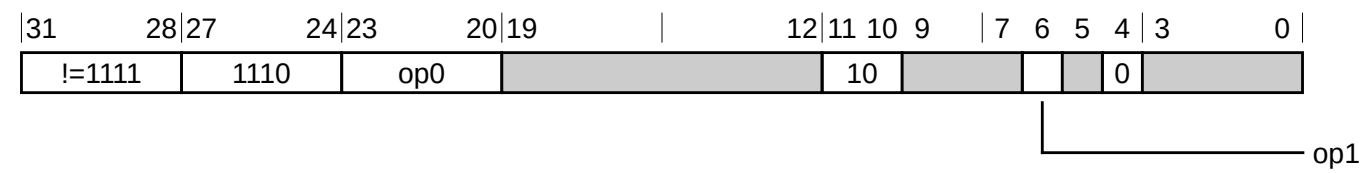


Table C4-61 Encoding table for the Floating-point data-processing group

Decode fields	Decode group or instruction page
op0	op1
1x11	1
1x11	0
!= 1x11	x

C4.1.17.1 Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from Floating-point data-processing.

31	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	10	9	8	7	6	5	4	3	0	
!=1111				1	1	1	0	1	D	1	1	o1	opc2		Vd		1	0	size	o3	1	M	0	Vm	
cond																									

Table C4-62

Decode fields				Instruction page
o1	opc2	size	o3	
x	xxx	0x	x	Unallocated.
0	000	10	0	VMOV (register) - Single-precision scalar variant
0	000	10	1	VABS - Single-precision scalar variant
0	000	11	0	VMOV (register) - Double-precision scalar variant
0	000	11	1	VABS - Double-precision scalar variant
0	001	10	0	VNEG - Single-precision scalar variant
0	001	10	1	VSQRT - Single-precision scalar variant
0	001	11	0	VNEG - Double-precision scalar variant
0	001	11	1	VSQRT - Double-precision scalar variant
0	010	01	x	Unallocated.
0	010	10	0	VCVTB - Half-precision to single-precision variant
0	010	10	1	VCVTT - Half-precision to single-precision variant
0	010	11	0	VCVTB - Half-precision to double-precision variant
0	010	11	1	VCVTT - Half-precision to double-precision variant
0	011	10	0	VCVTB - Single-precision to half-precision variant
0	011	10	1	VCVTT - Single-precision to half-precision variant
0	011	11	0	VCVTB - Double-precision to half-precision variant
0	011	11	1	VCVTT - Double-precision to half-precision variant
0	100	10	0	VCMP
0	100	10	1	VCMPE
0	100	11	0	VCMP
0	100	11	1	VCMPE
0	101	10	0	VCMP
0	101	10	1	VCMPE
0	101	11	0	VCMP
0	101	11	1	VCMPE
0	110	10	0	VRINTR - Single-precision scalar variant
0	110	10	1	VRINTZ (floating-point) - Single-precision scalar variant
0	110	11	0	VRINTR - Double-precision scalar variant
0	110	11	1	VRINTZ (floating-point) - Double-precision scalar variant

Decode fields			Instruction page	
o1	opc2	size	o3	
0	111	10	0	VRINTX (floating-point) - Single-precision scalar variant
0	111	10	1	VCVT (between double-precision and single-precision) - Single-precision to double-precision variant
0	111	11	0	VRINTX (floating-point) - Double-precision scalar variant
0	111	11	1	VCVT (between double-precision and single-precision) - Double-precision to single-precision variant
1	000	10	x	VCVT (integer to floating-point, floating-point) - Single-precision scalar variant
1	000	11	x	VCVT (integer to floating-point, floating-point) - Double-precision scalar variant
1	001	1x	x	Unallocated.
1	01x	10	x	VCVT (between floating-point and fixed-point, floating-point)
1	01x	11	x	VCVT (between floating-point and fixed-point, floating-point)
1	100	10	0	VCVTR
1	100	10	1	VCVT (floating-point to integer, floating-point)
1	100	11	0	VCVTR
1	100	11	1	VCVT (floating-point to integer, floating-point)
1	101	10	0	VCVTR
1	101	10	1	VCVT (floating-point to integer, floating-point)
1	101	11	0	VCVTR
1	101	11	1	VCVT (floating-point to integer, floating-point)
1	11x	10	x	VCVT (between floating-point and fixed-point, floating-point)
1	11x	11	x	VCVT (between floating-point and fixed-point, floating-point)

C4.1.17.2 Floating-point move immediate

This section describes the encoding of the Floating-point move immediate instruction class. The encodings in this section are decoded from [Floating-point data-processing](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0	
!=1111				1	1	1	0	1	D	1	1	imm4H		Vd		1	0	size	(0)	0	(0)	0	imm4L	
cond																								

Table C4-63

Decode fields	Instruction page
size	
0x	Unallocated.

Decode fields	Instruction page
size	
10	VMOV (immediate) - Single-precision scalar variant
11	VMOV (immediate) - Double-precision scalar variant

C4.1.17.3 Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from [Floating-point data-processing](#).

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0																				
!=1111				1		1		1		0		o0		D		o1		Vn				Vd				1		0		size		N		o2		M		0		Vm			
cond																																											

Table C4-64

Decode fields	Instruction page		
o0:o1	size	o2	
!= 111	0x	x	Unallocated.
000	10	0	VMLA (floating-point) - Single-precision scalar variant
000	10	1	VMLS (floating-point) - Single-precision scalar variant
000	11	0	VMLA (floating-point) - Double-precision scalar variant
000	11	1	VMLS (floating-point) - Double-precision scalar variant
001	10	0	VNMLS - Single-precision scalar variant
001	10	1	VNMLA - Single-precision scalar variant
001	11	0	VNMLS - Double-precision scalar variant
001	11	1	VNMLA - Double-precision scalar variant
010	10	0	VMUL (floating-point) - Single-precision scalar variant
010	10	1	VNMUL - Single-precision scalar variant
010	11	0	VMUL (floating-point) - Double-precision scalar variant
010	11	1	VNMUL - Double-precision scalar variant
011	10	0	VADD (floating-point) - Single-precision scalar variant
011	10	1	VSUB (floating-point) - Single-precision scalar variant
011	11	0	VADD (floating-point) - Double-precision scalar variant
011	11	1	VSUB (floating-point) - Double-precision scalar variant
100	10	0	VDIV - Single-precision scalar variant
100	11	0	VDIV - Double-precision scalar variant
101	10	0	VFNMS - Single-precision scalar variant
101	10	1	VFNMA - Single-precision scalar variant

Decode fields		Instruction page	
o0:o1	size	o2	
101	11	0	VFNMS - Double-precision scalar variant
101	11	1	VFNMA - Double-precision scalar variant
110	10	0	VFMA - Single-precision scalar variant
110	10	1	VFMS - Single-precision scalar variant
110	11	0	VFMA - Double-precision scalar variant
110	11	1	VFMS - Double-precision scalar variant

C4.1.18 Unconditional instructions

This section describes the encoding of the Unconditional instructions group. The encodings in this section are decoded from [A32 instruction set encoding](#).

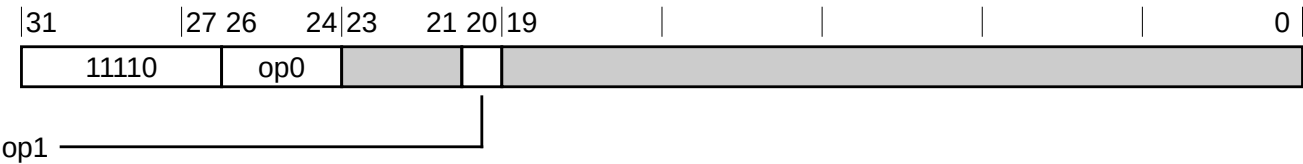


Table C4-65 Encoding table for the Unconditional instructions group

Decode fields		Decode group or instruction page
op0	op1	
00x	x	Miscellaneous
01x	x	Advanced SIMD data-processing
1xx	1	Memory hints and barriers
100	0	Advanced SIMD element or structure load/store
101	0	Unallocated.
11x	0	Unallocated.

C4.1.19 Miscellaneous

This section describes the encoding of the Miscellaneous group. The encodings in this section are decoded from [Unconditional instructions](#).



Table C4-66 Encoding table for the Miscellaneous group

Decode fields		Decode group or instruction page
op0	op1	
0xxxx	xxxx	Unallocated.
10000	xx0x	Change Process State
10010	0111	CONSTRAINED UNPREDICTABLE
10011	0111	Unallocated.
10010	!= 0111	Unallocated.
! = 100x0	0011	Unallocated.

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings](#).

C4.1.19.1 Change Process State

This section describes the encoding of the Change Process State instruction class. The encodings in this section are decoded from [Miscellaneous](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	0
1	1	1	1	0	0	0	1	0	0	0	0	imod	M	op	(0)	(0)	(0)	(0)	(0)	(0)	(0)	E	A	I	F	0	mode	

Table C4-67

Decode fields						Instruction page
imod	M	op	I	F	mode	
xx	x	1	0	0	0xxxx	SETEND
00	1	0	x	x	xxxxx	CPS, CPSID, CPSIE - Change mode variant
10	x	0	x	x	xxxxx	CPS, CPSID, CPSIE - Interrupt enable and change mode variant
xx	x	1	0	0	1xxxx	Unallocated.
xx	x	1	0	1	xxxxx	Unallocated.
xx	x	1	1	x	xxxxx	Unallocated.
11	x	0	x	x	xxxxx	CPS, CPSID, CPSIE - Interrupt disable and change mode variant

C4.1.20 Advanced SIMD data-processing

This section describes the encoding of the Advanced SIMD data-processing group. The encodings in this section are decoded from [Unconditional instructions](#).

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see [About the A32 Advanced SIMD and floating-point instructions and their encoding](#)

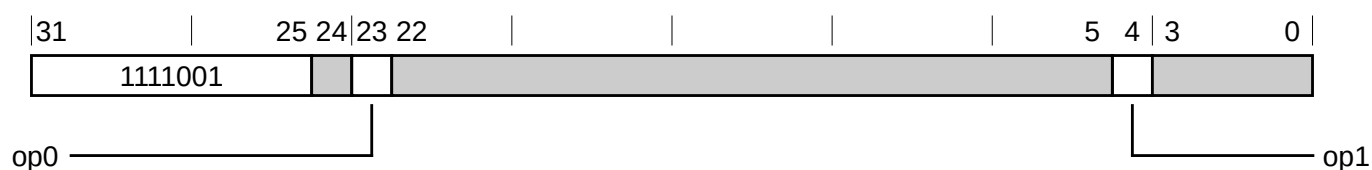


Table C4-68 Encoding table for the Advanced SIMD data-processing group

Decode fields		Decode group or instruction page
op0	op1	
0	x	Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

C4.1.20.1 Advanced SIMD three registers of the same length

This section describes the encoding of the Advanced SIMD three registers of the same length instruction class. The encodings in this section are decoded from [Advanced SIMD data-processing](#).

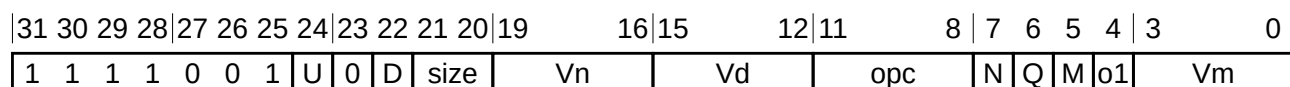


Table C4-69

Decode fields			Instruction page		
U	size	opc	Q	o1	
0	0x	1100	x	1	VFMA
0	0x	1101	x	0	VADD (floating-point)
0	0x	1101	x	1	VMLA (floating-point)
0	0x	1110	x	0	VCEQ (register) - A2
0	0x	1111	x	0	VMAX (floating-point)
0	0x	1111	x	1	VRECPS
x	xx	0000	x	0	VHADD
0	00	0001	x	1	VAND (register)

Decode fields					Instruction page
U	size	opc	Q	o1	
x	xx	0000	x	1	VQADD
x	xx	0001	x	0	VRHADD
0	00	1100	x	0	SHA1C
x	xx	0010	x	0	VHSUB
0	01	0001	x	1	VBIC (register)
x	xx	0010	x	1	VQSUB
x	xx	0011	x	0	VCGT (register) - A1
x	xx	0011	x	1	VCGE (register) - A1
0	01	1100	x	0	SHA1P
0	1x	1100	x	1	VFMS
0	1x	1101	x	0	VSUB (floating-point)
0	1x	1101	x	1	VMLS (floating-point)
0	1x	1110	x	0	Unallocated.
0	1x	1111	x	0	VMIN (floating-point)
0	1x	1111	x	1	VRSQRTS
x	xx	0100	x	0	VSHL (register)
0	xx	1000	x	0	VADD (integer)
0	10	0001	x	1	VORR (register)
0	xx	1000	x	1	VTST
x	xx	0100	x	1	VQSHL (register)
0	xx	1001	x	0	VMLA (integer)
x	xx	0101	x	0	VRSHL
x	xx	0101	x	1	VQRSHL
0	xx	1011	x	0	VQDMULH
0	10	1100	x	0	SHA1M
0	xx	1011	x	1	VPADD (integer)
x	xx	0110	x	0	VMAX (integer)
0	11	0001	x	1	VORN (register)
x	xx	0110	x	1	VMIN (integer)
x	xx	0111	x	0	VABD (integer)
x	xx	0111	x	1	VABA
0	11	1100	x	0	SHA1SU0
1	0x	1101	x	0	VPADD (floating-point)
1	0x	1101	x	1	VMUL (floating-point)
1	0x	1110	x	0	VCGE (register) - A2
1	0x	1110	x	1	VACGE
1	0x	1111	0	0	VPMAX (floating-point)
1	0x	1111	x	1	VMAXNM
1	00	0001	x	1	VEOR

Decode fields					Instruction page
U	size	opc	Q	o1	
x	xx	1001	x	1	VMUL (integer and polynomial)
1	00	1100	x	0	SHA256H
x	xx	1010	0	0	VPMAX (integer)
1	01	0001	x	1	VBSL
x	xx	1010	0	1	VPMIN (integer)
x	xx	1010	1	x	Unallocated.
1	01	1100	x	0	SHA256H2
1	1x	1101	x	0	VABD (floating-point)
1	1x	1110	x	0	VCGT (register) - A2
1	1x	1110	x	1	VACGT
1	1x	1111	0	0	VPMIN (floating-point)
1	1x	1111	x	1	VMINNM
1	xx	1000	x	0	VSUB (integer)
1	10	0001	x	1	VBIT
1	xx	1000	x	1	VCEQ (register) - A1
1	xx	1001	x	0	VMLS (integer)
1	1x	110x	x	0	Unallocated.
1	10	1100	x	0	SHA256SU1
1	11	0001	x	1	VBIF
1	xx	1111	1	0	Unallocated.

C4.1.21 Advanced SIMD two registers, or three registers of different lengths

This section describes the encoding of the Advanced SIMD two registers, or three registers of different lengths group. The encodings in this section are decoded from [Advanced SIMD data-processing](#).

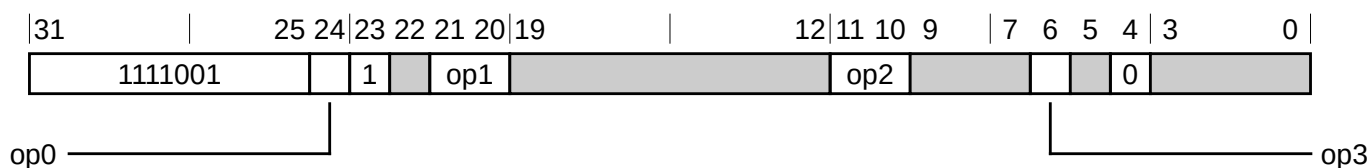


Table C4-70 Encoding table for the Advanced SIMD two registers, or three registers of different lengths group

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	11	xx	x	VEXT (byte elements)

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
1	11	0x	x	Advanced SIMD two registers misc
1	11	10	x	VTBL, VTBX
1	11	11	x	Advanced SIMD duplicate (scalar)
x	!= 11	xx	0	Advanced SIMD three registers of different lengths
x	!= 11	xx	1	Advanced SIMD two registers and a scalar

C4.1.21.1 Advanced SIMD two registers misc

This section describes the encoding of the Advanced SIMD two registers misc instruction class. The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	7	6	5	4	3	0
1	1	1	1	0	0	1	1	1	D	1	1	size	opc1	Vd	0	opc2	Q	M	0	Vm					

Table C4-71

Decode fields				Instruction page	Feature
size	opc1	opc2	Q		
xx	00	0000	x	VREV64	-
xx	00	0001	x	VREV32	-
xx	00	0010	x	VREV16	-
xx	00	0011	x	Unallocated.	-
xx	00	010x	x	VPADDL	-
xx	00	0110	0	AESE	FEAT_AES
xx	00	0110	1	AESD	FEAT_AES
xx	00	0111	0	AESMC	FEAT_AES
xx	00	0111	1	AESIMC	FEAT_AES
xx	00	1000	x	VCLS	-
00	10	0000	x	VSWP	-
xx	00	1001	x	VCLZ	-
xx	00	1010	x	VCNT	-
xx	00	1011	x	VMVN (register)	-
00	10	1100	1	Unallocated.	-
xx	00	110x	x	VPADAL	-
xx	00	1110	x	VQABS	-
xx	00	1111	x	VQNEG	-

Decode fields				Instruction page	Feature
size	opc1	opc2	Q		
xx	01	x000	x	VCGT (immediate #0)	-
xx	01	x001	x	VCGE (immediate #0)	-
xx	01	x010	x	VCEQ (immediate #0)	-
xx	01	x011	x	VCLE (immediate #0)	-
xx	01	x100	x	VCLT (immediate #0)	-
xx	01	x110	x	VABS	-
xx	01	x111	x	VNEG	-
xx	01	0101	1	Unallocated.	-
xx	01	0101	1	SHA1H	FEAT_SHA1
01	10	11x0	1	Unallocated.	-
xx	10	0001	x	VTRN	-
xx	10	0010	x	VUZP	-
xx	10	0011	x	VZIP	-
xx	10	0100	0	VMOVN	-
xx	10	0100	1	VQMOVN, VQMOVUN - Unsigned result variant	-
xx	10	0101	x	VQMOVN, VQMOVUN - Signed result variant	-
xx	10	0110	0	VSHLL	-
xx	10	0111	0	SHA1SU1	FEAT_SHA1
xx	10	0111	1	SHA256SU0	FEAT_SHA256
xx	10	1000	x	VRINTN (Advanced SIMD)	-
xx	10	1001	x	VRINTX (Advanced SIMD)	-
xx	10	1010	x	VRINTA (Advanced SIMD)	-
xx	10	1011	x	VRINTZ (Advanced SIMD)	-
10	10	1100	1	Unallocated.	-
xx	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Single-precision to half-precision variant	-
xx	10	1101	x	VRINTM (Advanced SIMD)	-
xx	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) - Half-precision to single-precision variant	-
xx	10	1110	1	Unallocated.	-
xx	10	1111	x	VRINTP (Advanced SIMD)	-
xx	11	000x	x	VCVTA (Advanced SIMD)	-
xx	11	001x	x	VCVTN (Advanced SIMD)	-
xx	11	010x	x	VCVTP (Advanced SIMD)	-
xx	11	011x	x	VCVTM (Advanced SIMD)	-
xx	11	10x0	x	VRECPE	-
xx	11	10x1	x	VRSQRTE	-
11	10	1100	1	Unallocated.	-
xx	11	11xx	x	VCVT (between floating-point and integer, Advanced SIMD)	-

C4.1.21.2 Advanced SIMD duplicate (scalar)

This section describes the encoding of the Advanced SIMD duplicate (scalar) instruction class. The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

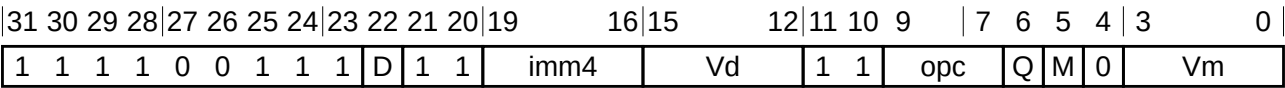


Table C4-72

Decode fields	Instruction page
opc	
000	VDUP (scalar)
001	Unallocated.
01x	Unallocated.
1xx	Unallocated.

C4.1.21.3 Advanced SIMD three registers of different lengths

This section describes the encoding of the Advanced SIMD three registers of different lengths instruction class. The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

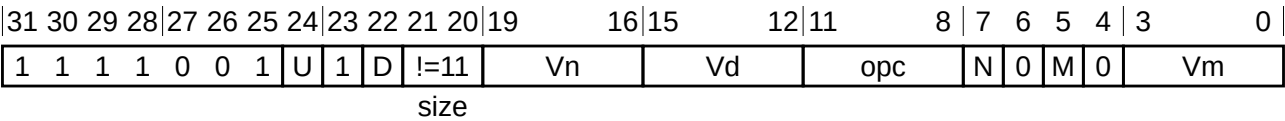


Table C4-73

Decode fields		Instruction page
U	opc	
x	0000	VADDL
x	0001	VADDW
x	0010	VSUBL
0	0100	VADDHN
x	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL

Decode fields		Instruction page
U	opc	
x	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
x	0111	VABDL (integer)
x	1000	VMLAL (integer)
x	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
x	11x0	VMULL (integer and polynomial)
1	1001	Unallocated.
1	1011	Unallocated.
1	1101	Unallocated.
x	1111	Unallocated.

C4.1.21.4 Advanced SIMD two registers and a scalar

This section describes the encoding of the Advanced SIMD two registers and a scalar instruction class. The encodings in this section are decoded from [Advanced SIMD two registers, or three registers of different lengths](#).

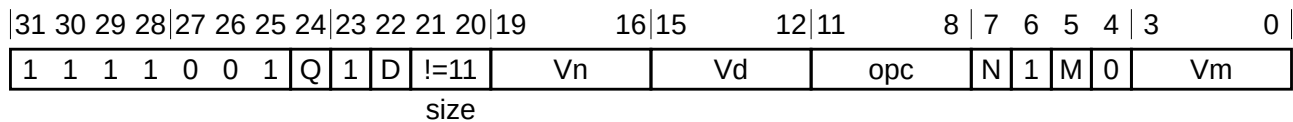


Table C4-74

Decode fields		Instruction page
Q	opc	
x	000x	VMLA (by scalar)
0	0011	VQDMLAL
x	0010	VMLAL (by scalar)
0	0111	VQDMLSL
x	010x	VMLS (by scalar)
0	1011	VQDMULL
x	0110	VMLSL (by scalar)
x	100x	VMUL (by scalar)
1	0011	Unallocated.
x	1010	VMULL (by scalar)

Decode fields		Instruction page
Q	opc	
1	0111	Unallocated.
x	1100	VQDMULH
x	1101	VQRDMULH
1	1xx1	Unallocated.

C4.1.22 Advanced SIMD shifts and immediate generation

This section describes the encoding of the Advanced SIMD shifts and immediate generation group. The encodings in this section are decoded from [Advanced SIMD data-processing](#).

31	25	24	23	22	21					7	6	5	4	3	0
1111001		1						op0			1				

Table C4-75 Encoding table for the Advanced SIMD shifts and immediate generation group

Decode fields	Decode group or instruction page
op0	
000xxxxxxxxxxx0	Advanced SIMD one register and modified immediate
!= 000xxxxxxxxxxx0	Advanced SIMD two registers and shift amount

C4.1.22.1 Advanced SIMD one register and modified immediate

This section describes the encoding of the Advanced SIMD one register and modified immediate instruction class. The encodings in this section are decoded from [Advanced SIMD shifts and immediate generation](#).

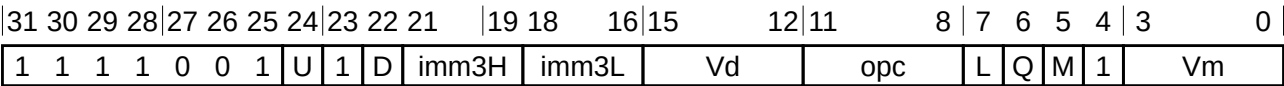
31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3	Vd	cmode	0	Q	op	1	imm4				

Table C4-76

Decode fields		Instruction page
cmode	op	
0xx0	0	VMOV (immediate) - A1
0xx0	1	VMVN (immediate) - A1
0xx1	0	VORR (immediate) - A1
0xx1	1	VBIC (immediate) - A1
10x0	0	VMOV (immediate) - A3
10x0	1	VMVN (immediate) - A2
10x1	0	VORR (immediate) - A2
10x1	1	VBIC (immediate) - A2
11xx	0	VMOV (immediate) - A4
110x	1	VMVN (immediate) - A3
1110	1	VMOV (immediate) - A5
1111	1	Unallocated.

C4.1.22.2 Advanced SIMD two registers and shift amount

This section describes the encoding of the Advanced SIMD two registers and shift amount instruction class. The encodings in this section are decoded from [Advanced SIMD shifts and immediate generation](#).



This decode also imposes the constraint:

- !(imm3H == '000' && L == '0').

Table C4-77

Decode fields						Instruction page
U	imm3H	imm3L	opc	L	Q	
x	xxx	xxx	1xxx	1	x	Unallocated.
x	000	xxx	0000	1	0	VSHR 64-bit SIMD vector
x	000	xxx	0000	1	1	VSHR 128-bit SIMD vector
x	000	xxx	0001	1	0	VSRA 64-bit SIMD vector
x	000	xxx	0001	1	1	VSRA 128-bit SIMD vector
x	000	xxx	0010	1	0	VRSHR 64-bit SIMD vector
x	000	xxx	0010	1	1	VRSHR 128-bit SIMD vector

Decode fields						Instruction page
U	imm3H	imm3L	opc	L	Q	
x	000	xxx	0011	1	0	VRSRA 64-bit SIMD vector
x	000	xxx	0011	1	1	VRSRA 128-bit SIMD vector
x	000	xxx	0111	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
x	000	xxx	0111	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
x	001	000	1010	0	0	VMOVL
x	001	!= 000	1010	0	0	VSHLL
x	010	000	1010	0	0	VMOVL
x	010	!= 000	1010	0	0	VSHLL
x	011	xxx	1010	0	0	VSHLL
x	100	000	1010	0	0	VMOVL
x	100	!= 000	1010	0	0	VSHLL
x	101	xxx	1010	0	0	VSHLL
x	11x	xxx	1010	0	0	VSHLL
x	!= 000	xxx	0000	x	0	VSHR 64-bit SIMD vector
x	!= 000	xxx	0000	x	1	VSHR 128-bit SIMD vector
x	!= 000	xxx	0001	x	0	VSRA 64-bit SIMD vector
x	!= 000	xxx	0001	x	1	VSRA 128-bit SIMD vector
x	!= 000	xxx	0010	x	0	VRSHR 64-bit SIMD vector
x	!= 000	xxx	0010	x	1	VRSHR 128-bit SIMD vector
x	!= 000	xxx	0011	x	0	VRSRA 64-bit SIMD vector
x	!= 000	xxx	0011	x	1	VRSRA 128-bit SIMD vector
x	!= 000	xxx	0111	x	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, signed result
x	!= 000	xxx	0111	x	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, signed result
x	!= 000	xxx	1001	0	0	VQSHRN, VQSHRUN — Signed result
x	!= 000	xxx	1001	0	1	VQRSHRN, VQRSHRUN — Signed result
x	!= 000	xxx	1010	0	1	Unallocated.
x	!= 000	xxx	1011	0	x	Unallocated.
x	!= 000	xxx	11xx	0	0	VCVT (between floating-point and fixed-point, Advanced SIMD) — 64-bit SIMD vector
x	!= 000	xxx	11xx	0	1	VCVT (between floating-point and fixed-point, Advanced SIMD) — 128-bit SIMD vector
0	xxx	xxx	01x0	1	x	Unallocated.
0	000	xxx	0101	1	0	VSHL (immediate) — 64-bit SIMD vector
0	000	xxx	0101	1	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000	xxx	01x0	0	x	Unallocated.
0	!= 000	xxx	0101	x	0	VSHL (immediate) — 64-bit SIMD vector
0	!= 000	xxx	0101	x	1	VSHL (immediate) — 128-bit SIMD vector
0	!= 000	xxx	1000	0	0	VSHRN
0	!= 000	xxx	1000	0	1	VRSHRN
1	000	xxx	0100	1	0	VSRI — 64-bit SIMD vector

Decode fields						Instruction page
U	imm3H	imm3L	opc	L	Q	
1	000	xxx	0100	1	1	VSRI — 128-bit SIMD vector
1	000	xxx	0101	1	0	VSLI — 64-bit SIMD vector
1	000	xxx	0101	1	1	VSLI — 128-bit SIMD vector
1	000	xxx	0110	1	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	000	xxx	0110	1	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000	xxx	0100	x	0	VSRI — 64-bit SIMD vector
1	!= 000	xxx	0100	x	1	VSRI — 128-bit SIMD vector
1	!= 000	xxx	0101	x	0	VSLI — 64-bit SIMD vector
1	!= 000	xxx	0101	x	1	VSLI — 128-bit SIMD vector
1	!= 000	xxx	0110	x	0	VQSHL, VQSHLU (immediate) — 64-bit SIMD vector, unsigned result
1	!= 000	xxx	0110	x	1	VQSHL, VQSHLU (immediate) — 128-bit SIMD vector, unsigned result
1	!= 000	xxx	1000	0	0	VQSHRN, VQSHRUN — Unsigned result
1	!= 000	xxx	1000	0	1	VQRSHRN, VQRSHRUN — Unsigned result

C4.1.23 Memory hints and barriers

This section describes the encoding of the Memory hints and barriers group. The encodings in this section are decoded from [Unconditional instructions](#).

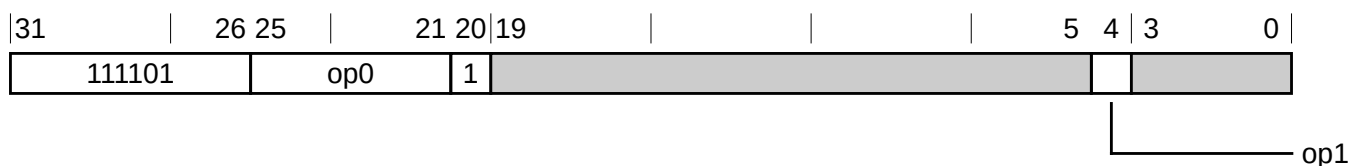


Table C4-78 Encoding table for the Memory hints and barriers group

Decode fields		Decode group or instruction page
op0	op1	
00xx1	x	CONSTRAINED UNPREDICTABLE
01001	x	CONSTRAINED UNPREDICTABLE
01011	x	Barriers
011x1	x	CONSTRAINED UNPREDICTABLE
0xxx0	x	Preload (immediate)
1xxx0	0	Preload (register)
1xxx1	0	CONSTRAINED UNPREDICTABLE
1xxxx	1	Unallocated.

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings](#).

C4.1.23.1 Barriers

This section describes the encoding of the Barriers instruction class. The encodings in this section are decoded from [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	4	3	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	opcode		option	

Table C4-79

Decode fields		Instruction page
opcode	option	
0000	xxxx	CONSTRAINED UNPREDICTABLE
00x0	xxxx	CONSTRAINED UNPREDICTABLE
0001	xxxx	CLREX
0100	0000	CONSTRAINED UNPREDICTABLE
0100	!= 0x00	DSB
0101	xxxx	DMB
0110	xxxx	ISB
1xxx	xxxx	CONSTRAINED UNPREDICTABLE

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings](#).

C4.1.23.2 Preload (immediate)

This section describes the encoding of the Preload (immediate) instruction class. The encodings in this section are decoded from [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	14	13	12	11								0
1	1	1	1	0	1	0	D	U	R	0	1	Rn	(1)	(1)	(1)	(1)	imm12									

Table C4-80

Decode fields			Instruction page
D	R	Rn	
0	0	xxxx	Reserved hint, behaves as NOP.
0	1	xxxx	PLI (immediate, literal)
1	x	1111	PLD (literal)
1	0	!= 1111	PLD, PLDW (immediate) - Preload write variant
1	1	!= 1111	PLD, PLDW (immediate) - Preload read variant

C4.1.23.3 Preload (register)

This section describes the encoding of the Preload (register) instruction class. The encodings in this section are decoded from [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	14	13	12	11	7	6	5	4	3	0
1	1	1	1	0	1	1	D	U	o2	0	1	Rn	(1)	(1)	(1)	(1)	imm5	styp	0	Rm				

Table C4-81

Decode fields			Instruction page
D	o2	imm5:styp	
0	0	xxxxxxxx	Reserved hint, behaves as NOP.
0	1	!= 0000011	PLI (register) - Shift or rotate by value variant
0	1	0000011	PLI (register) - Rotate right with extend variant
1	0	!= 0000011	PLD, PLDW (register) - Preload write, optional shift or rotate variant
1	0	0000011	PLD, PLDW (register) - Preload write, rotate right with extend variant
1	1	!= 0000011	PLD, PLDW (register) - Preload read, optional shift or rotate variant
1	1	0000011	PLD, PLDW (register) - Preload read, rotate right with extend variant

C4.1.24 Advanced SIMD element or structure load/store

This section describes the encoding of the Advanced SIMD element or structure load/store group. The encodings in this section are decoded from [Unconditional instructions](#).

This group has encodings in both the T32 and A32 instruction sets. For information about mappings between the encodings of this group, see [About the A32 Advanced SIMD and floating-point instructions and their encoding](#)

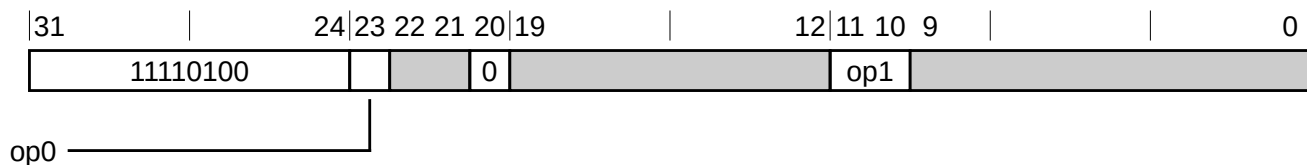


Table C4-82 Encoding table for the Advanced SIMD element or structure load/store group

Decode fields		Decode group or instruction page
op0	op1	
0	xx	Advanced SIMD load/store multiple structures
1	11	Advanced SIMD load single structure to all lanes
1	!= 11	Advanced SIMD load/store single structure to one lane

C4.1.24.1 Advanced SIMD load/store multiple structures

This section describes the encoding of the Advanced SIMD load/store multiple structures instruction class. The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

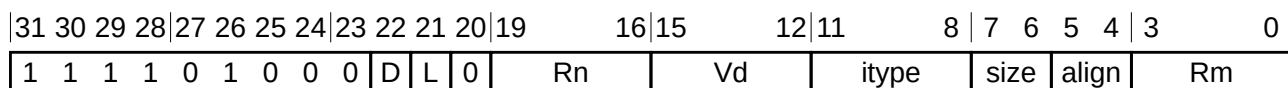


Table C4-83

Decode fields			Instruction page
L	itype	Rm	
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)

Decode fields			Instruction page
L	itype	Rm	
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
x	1011	xxxx	Unallocated.
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
x	11xx	xxxx	Unallocated.
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)

Decode fields			Instruction page
L	itype	Rm	
1	1010	1111	VLD1 (multiple single elements)

C4.1.24.2 Advanced SIMD load single structure to all lanes

This section describes the encoding of the Advanced SIMD load single structure to all lanes instruction class. The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	1	0	1	0	0	1	D	L	0	Rn	Vd	1	1	N	size	T	a	Rm					

Table C4-84

Decode fields				Instruction page
L	N	a	Rm	
0	xx	x	xxxx	Unallocated.
1	00	x	!= 11x1	VLD1 (single element to all lanes)
1	00	x	1101	VLD1 (single element to all lanes)
1	00	x	1111	VLD1 (single element to all lanes)
1	01	x	!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01	x	1101	VLD2 (single 2-element structure to all lanes)
1	01	x	1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1	xxxx	Unallocated.
1	11	x	!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11	x	1101	VLD4 (single 4-element structure to all lanes)
1	11	x	1111	VLD4 (single 4-element structure to all lanes)

C4.1.24.3 Advanced SIMD load/store single structure to one lane

This section describes the encoding of the Advanced SIMD load/store single structure to one lane instruction class. The encodings in this section are decoded from [Advanced SIMD element or structure load/store](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	4	3	0
1	1	1	1	0	1	0	0	1	D	L	0	Rn	Vd	!=11	N	index_align	size	Rm					

Table C4-85

Decode fields				Instruction page
L	size	N	Rm	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)

Decode fields				Instruction page
L	size	N	Rm	
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

C4.2 About the A32 Advanced SIMD and floating-point instructions and their encoding

The Advanced SIMD and floating-point instructions are common to the T32 and A32 instruction sets. These instructions perform Advanced SIMD and floating-point operations on a common register file, the SIMD&FP register file. This means:

- In general, the instructions that load or store registers in this file, or move data between general-purpose registers and this register file, are common to the Advanced SIMD and floating-point instructions.
- There are distinct Advanced SIMD data-processing instructions and floating-point data-processing instructions.

All A32 Advanced SIMD and floating-point instructions have 32-bit encodings. Different groups of these instructions are decoded from different points in the 32-bit A32 instruction decode structure. [Table C4-86](#) shows these instruction groups, and where each group is decoded from the overall A32 decode structure:

Table C4-86 Advanced SIMD and floating-point instructions in the A32 decode structure

Advanced SIMD and floating-point instruction group	A32 decode is from
Advanced SIMD and System register load/store and 64-bit move	System register access, Advanced SIMD, floating-point, and Supervisor call
Floating-point data-processing	System register access, Advanced SIMD, floating-point, and Supervisor call
Advanced SIMD and System register 32-bit move	System register access, Advanced SIMD, floating-point, and Supervisor call
Advanced SIMD data-processing	Unconditional instructions
Advanced SIMD element or structure load/store	Unconditional instructions

Chapter C5

T32 and A32 Base Instruction Set Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- [Alphabetical list of T32 and A32 base instruction set instructions](#)
- [Encoding and use of banked register transfer instructions](#)

C5.1 Alphabetical list of T32 and A32 base instruction set instructions

This section lists every instruction in the T32 and A32 base instruction sets. For details of the format used see [Format of instruction descriptions](#).

This section is formatted so that each instruction description starts on a new page.

C5.1.1 ADC, ADCS (immediate)

Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register.

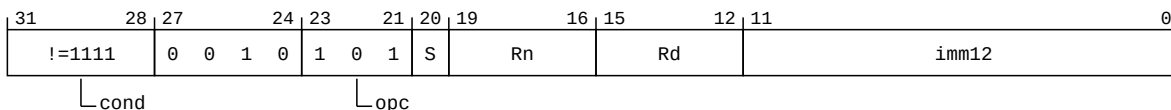
If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the ADC variant

Applies when (S == 0)

```
ADC{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Encoding for the ADCS variant

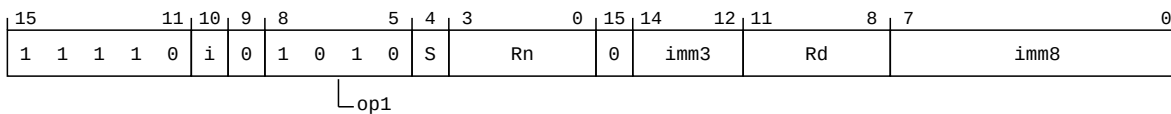
Applies when (S == 1)

```
ADCS{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1



Encoding for the ADC variant

Applies when (S == 0)

```
ADC{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Encoding for the ADCS variant

Applies when (S == 1)

```
ADCS{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 ADC” and “A1 ADCS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ADCS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the “T1 ADC” and “T1 ADCS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 ADC” and “A1 ADCS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 ADC” and “T1 ADCS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 ADC” and “A1 ADCS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1 ADC” and “T1 ADCS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

C5.1.2 ADC, ADCS (register)

Add with Carry (register) adds a register value, the Carry flag value, and an optionally-shifted register value, and writes the result to the destination register.

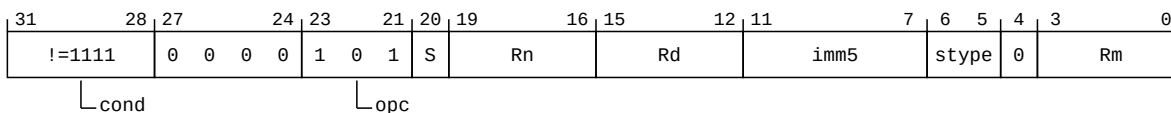
If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <RD> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the ADC, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

ADC{<c>}{<q>}{<Rd>}{<Rn>}{<Rm>}{RRX}

Encoding for the ADC, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

```
ADC{<c>}{<q>}{<Rd>,<Rn>,<Rm>,<shift>#<amount>}
```

Encoding for the ADCS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

ADCS { <c> } { <q> } { <Rd>, } <Rn>, <Rm>, RRX

Encoding for the ADCS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

ADCS{<c>}{<q>}{<Rd>,<Rn>,<Rm>{,<shift>#<amount>}

Decode for all variants of this encoding

```

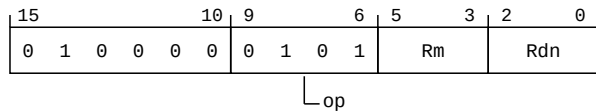
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

let setflags : boolean = (S == '1');

let (shift t, shift n) : (SRTyp, integer) = DecodeImmShift(stype, imm5);

```

T1



Encoding

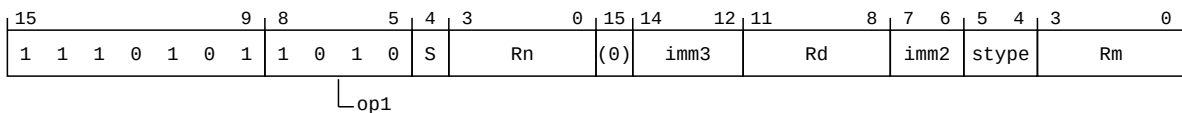
```
ADC<c>{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

ADCS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the ADC, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

ADC {<c>} {<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADC, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ADC{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}
```

ADC<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the ADCS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

ADCS {<c>} {<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADCS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ADCS{<c>}{<q>}{<Rd>,<Rn>,<Rm>{,<shift>#<amount>}

ADCS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 ADC, rotate right with extend”, “A1 ADC, shift or rotate by value”, “A1 ADCS, rotate right with extend”, and “A1 ADCS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ADCS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T2 ADC, rotate right with extend”, “T2 ADC, shift or rotate by value”, “T2 ADCS, rotate right with extend”, and “T2 ADCS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 ADC, rotate right with extend”, “A1 ADC, shift or rotate by value”, “A1 ADCS, rotate right with extend”, and “A1 ADCS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 ADC, rotate right with extend”, “T2 ADC, shift or rotate by value”, “T2 ADCS, rotate right with extend”, and “T2 ADCS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 ADC, rotate right with extend”, “A1 ADC, shift or rotate by value”, “A1 ADCS, rotate right with extend”, and “A1 ADCS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 ADC, rotate right with extend”, “T2 ADC, shift or rotate by value”, “T2 ADCS, rotate right with extend”, and “T2 ADCS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

type	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 ADC, shift or rotate by value” and “A1 ADCS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 ADC, shift or rotate by value” and “T2 ADCS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

In T32 assembly:

- Outside an IT block, if ADCS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADCS <Rd>, <Rn> had been written.
- Inside an IT block, if ADC<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADC<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation

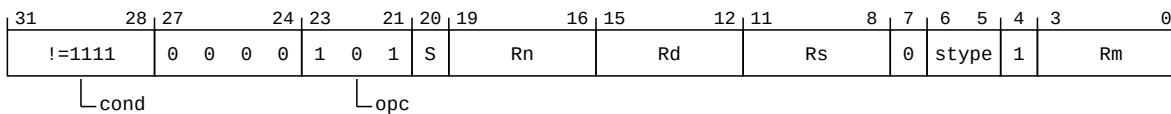
```

if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcv;
        end;
    end;
end;
end;

```

C5.1.3 ADC, ADCS (register-shifted register)

Add with Carry (register-shifted register) adds a register value, the Carry flag value, and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

ADCS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

```
ADC{<c>}{<q>}{<Rd>,<Rn>,<Rm>,<shift><Rs>}
```

Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

 $\langle R_n \rangle$

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

C5.1.4 ADD (immediate, to PC)

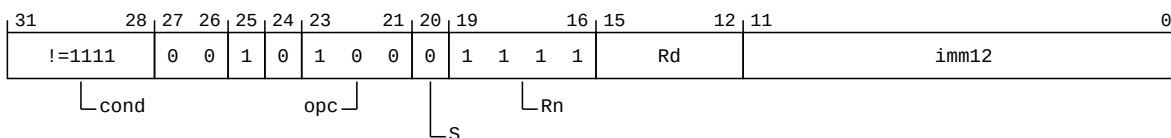
Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This is a pseudo-instruction of [ADR](#). This means:

- The encodings in this description are named to match the encodings of [ADR](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [ADR](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T3](#)).

A1



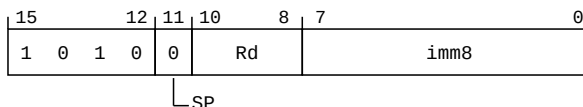
Encoding for the A1 variant

ADD{<c>}{<q>} <Rd>, PC, #<const>

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

T1



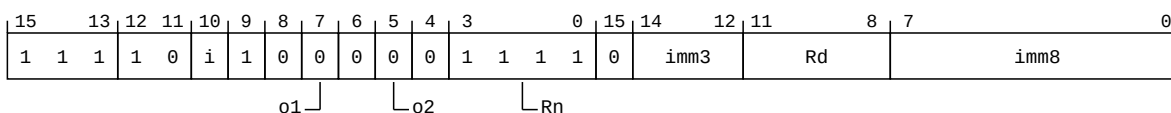
Encoding for the T1 variant

ADD{<c>}{<q>} <Rd>, PC, #<imm8>

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

T3



Encoding for the T3 variant

ADD{<c>}{<q>} <Rd>, PC, #<imm12>

ADDW{<c>}{<q>} <Rd>, PC, #<imm12> // (UInt(Rd) < 8 && UInt(i :: imm3 :: imm8) < 256)

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1” variant: is the general-purpose destination register, encoded in the “Rd” field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T1” and “T3” variants: is the general-purpose destination register, encoded in the “Rd” field.

<const>

An immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

<imm8>

Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the “imm8” field as <imm8>/4.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

Operation

The description of [ADR](#) gives the operational pseudocode for this instruction.

C5.1.5 ADD, ADDS (immediate)

Add (immediate) adds an immediate value to a register value, and writes the result to the destination register.

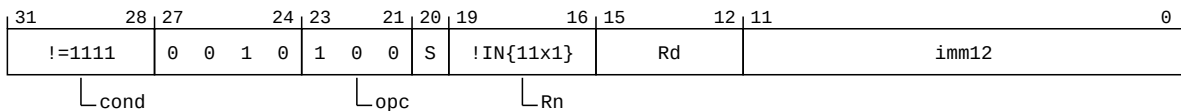
If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), [T3](#), and [T4](#)).

A1



Encoding for the ADD variant

Applies when (S == 0 && Rn != 11x1)

ADD{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the ADDS variant

Applies when (S == 1 && Rn != 1101)

ADDS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

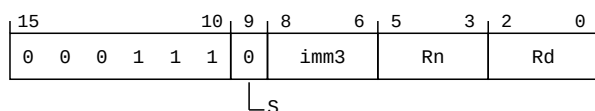
Decode for all variants of this encoding

```

if Rn == '1111' && S == '0' then See("ADR"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);

```

T1



Encoding

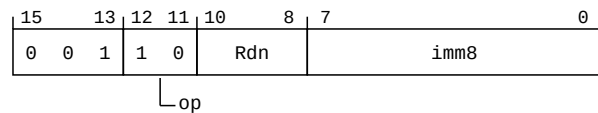
```
ADD<c>{<q>} <Rd>, <Rn>, #<imm3> // (InITBlock())
```

```
ADDS{<q>} <Rd>, <Rn>, #<imm3> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm3);
```

T2



Encoding

```
ADD<c>{<q>} <Rdn>, #<imm8> // (InITBlock() && UInt(imm8) < 8)
```

```
ADD<c>{<q>} {<Rdn>, }<Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)
```

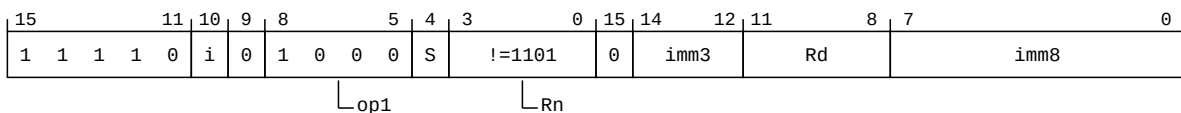
```
ADDS{<q>} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)
```

```
ADDS{<q>} {<Rdn>, }<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm8);
```

T3



Encoding for the ADD variant

Applies when (S == 0)

```
ADD{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

```
ADD<c>.W {<Rd>, }<Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1
↳ or T2)
```

Encoding for the ADDS variant

Applies when (S == 1 && Rd != 1111)

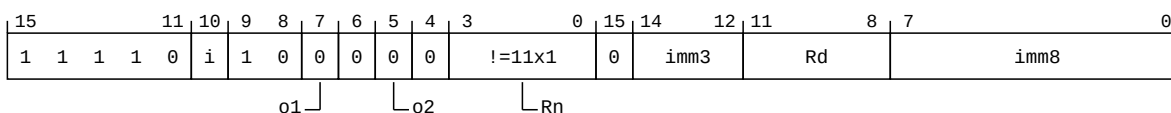
ADDS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

ADDS.W {<Rd>, }<Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
↪T2)

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (immediate)"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

T4



Encoding

ADD{<c>}{<q>} {<Rd>, }<Rn>, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

ADDW{<c>}{<q>} {<Rd>, }<Rn>, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

Decode for this encoding

```
if Rn == '1111' then See("ADR"); end;
if Rn == '1101' then See("ADD (SP plus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{i::imm3::imm8};
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 ADD” and “A1 ADDS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ADDS variant, the instruction performs an exception return, that restores `PSTATE` from `SPSR_<current_mode>`. Arm deprecates use of this instruction.

For the “T1”, “T3 ADD”, “T3 ADDS”, and “T4” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 ADD”, “A1 ADDS”, and “T4” variants: is the general-purpose source register, encoded in the “Rn” field. If the SP is used, see [ADD \(SP plus immediate\)](#). If the PC is used, see [ADR](#).

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

For the “T3 ADD” and “T3 ADDS” variants: is the general-purpose source register, encoded in the “Rn” field. If the SP is used, see [ADD \(SP plus immediate\)](#).

<const>

For the “A1 ADD” and “A1 ADDS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T3 ADD” and “T3 ADDS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

<imm3>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “imm3” field.

<Rdn>

Is the general-purpose source and destination register, encoded in the “Rdn” field.

<imm8>

Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `ADDW` syntax). Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let (result, nzcvt) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
        if d == 15 then           // Can only occur for A32 encoding
            if setflags then
                ALUEXCEPTIONReturn(result);
            else
                ALUWritePC(result);
            end;
        else
            R(d) = result;
            if setflags then
                PSTATE.[N,Z,C,V] = nzcvt;
            end;
        end;
    end;
end;
else
```

```
    if ConditionPassed() then
        EncodingSpecificOperations();
        let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

C5.1.6 ADD, ADDS (register)

Add (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

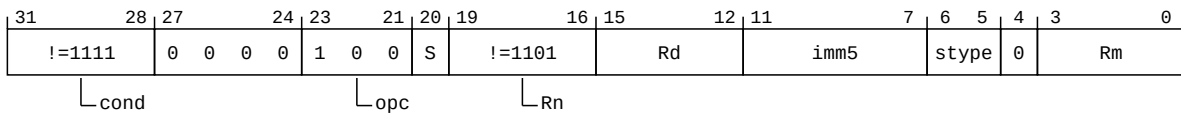
If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the ADD, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

ADD{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADD, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

ADD{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the ADDS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

ADDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADDS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

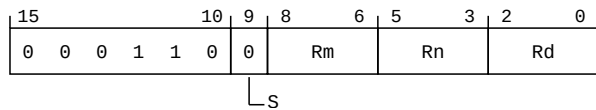
ADDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```

if Rn == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
    
```

T1



Encoding

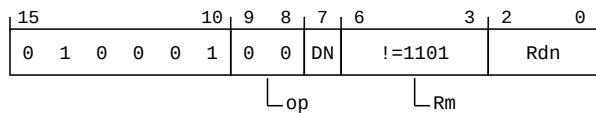
ADD<c>{<q>} <Rd>, <Rn>, <Rm> // (InITBlock())

ADDS{<q>} {<Rd>, }<Rn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

Applies when (!(DN == '1' && Rdn == '101'))

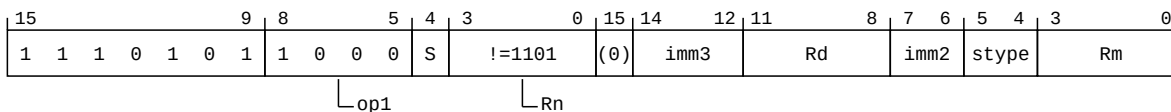
ADD{<c>}{<q>} {<Rdn>, }<Rdn>, <Rm>

ADD<c>{<q>} <Rdn>, <Rm> // (Preferred syntax, Inside IT block)

Decode for this encoding

```
if (DN::Rdn == '1101' || Rm == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(DN::Rdn);
let n : integer = d;
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if n == 15 && m == 15 then UnpredictableProcedure(); end;
if d == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T3



Encoding for the ADD, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

ADD{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADD, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ADD{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

ADD<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

ADD{<c>}.W {<Rd>, }<Rn>, <Rm> // (<Rd> == <Rn>, and <Rd>, <Rn>, <Rm> can be represented in T2)

Encoding for the ADDS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

ADDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ADDS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ADDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

ADDS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2)

Decode for all variants of this encoding

```

if Rd == '1111' && S == '1' then See("CMN (register)"); end;
if Rn == '1101' then See("ADD (SP plus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, and “A1 ADDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. If the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ADDS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`. Arm deprecates use of this instruction.

For the “T1” variant: is the general-purpose destination register, encoded in the “Rd” field.

When used inside an IT block, <Rd> must be specified. When used outside an IT block, <Rd> is optional, and:

- If omitted, this register is the same as <Rn>.
- If present, encoding T1 is preferred to encoding T2.

For the “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, and “A1 ADDS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used. If the SP is used, see [ADD \(SP plus register\)](#).

For the “T1” variant: is the first general-purpose source register, encoded in the “Rn” field.

For the “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. If the SP is used, see [ADD \(SP plus register\)](#).

<Rm>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, and “A1 ADDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

For the “T2” variant: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 ADD, shift or rotate by value” and “A1 ADDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T3 ADD, shift or rotate by value” and “T3 ADDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the general-purpose source and destination register, encoded in the “DN:Rdn” field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

The assembler language allows <Rdn> to be specified once or twice in the assembler syntax. When used inside an IT block, and <Rdn> and <Rm> are in the range R0 to R7, <Rdn> must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when <Rdn> is specified once or twice.

Inside an IT block, if ADD<c> <Rd>, <Rn>, <Rd> cannot be assembled using encoding T1, it is assembled using encoding T2 as though ADD<c> <Rd>, <Rn> had been written. To prevent this happening, use the .w qualifier.

Operation

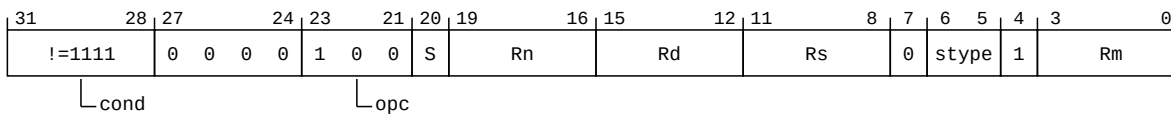
```

if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    if d == 15 then
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
end;

```

C5.1.7 ADD, ADDS (register-shifted register)

Add (register-shifted register) adds a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

ADDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

ADD{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

C5.1.8 ADD, ADDS (SP plus immediate)

Add to SP (immediate) adds an immediate value to the SP value, and writes the result to the destination register.

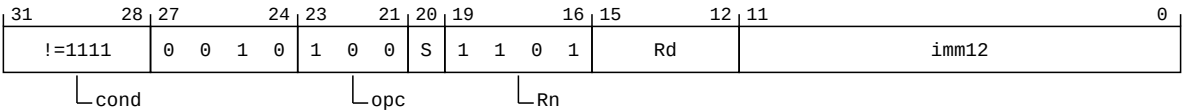
If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADDS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores `PSTATE` from `SPSR_<current_mode>`.
 - The PE checks `SPSR_<current_mode>` for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), [T3](#), and [T4](#)).

A1



Encoding for the ADD variant

Applies when (S == 0)

```
ADD{<c>}{<q>} {<Rd>, }SP, #<const>
```

Encoding for the ADDS variant

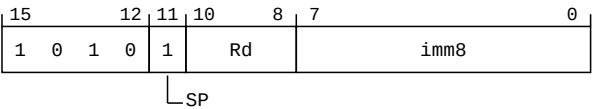
Applies when (S == 1)

```
ADDS{<c>}{<q>} {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1



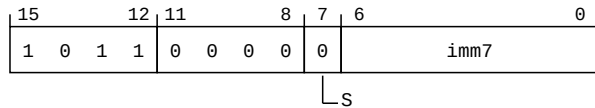
Encoding

```
ADD{<c>}{<q>} <Rd>, SP, #<imm8>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
```

T2



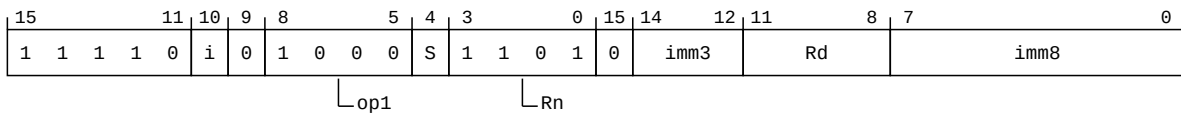
Encoding

```
ADD{<c>}{<q>} {SP, }SP, #<imm7>
```

Decode for this encoding

```
let d : integer = 13;
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm7::'00');
```

T3



Encoding for the ADD variant

Applies when (S == 0)

```
ADD{<c>}{<q>} {<Rd>, }SP, #<const>
```

```
ADD{<c>}.W {<Rd>, }SP, #<const> // (<Rd>, <const> can be represented in T1 or T2)
```

Encoding for the ADDS variant

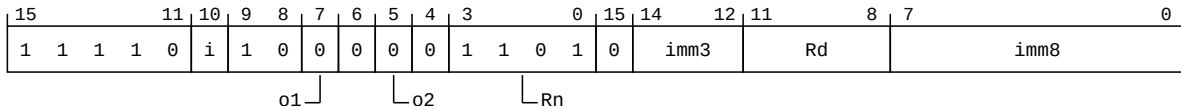
Applies when (S == 1 && Rd != 1111)

```
ADDS{<c>}{<q>} {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (immediate)"); end;
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if d == 15 && !setflags then UnpredictableProcedure(); end;
```

T4



Encoding

ADD{<c>}{<q>} {<Rd>, }SP, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

ADDW{<c>}{<q>} {<Rd>, }SP, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{(i::imm3::imm8)};
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 ADD” and “A1 ADDS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ADDS variant, the instruction performs an exception return, that restores `PSTATE` from `SPSR_<current_mode>`.

For the “T1” variant: is the general-purpose destination register, encoded in the “Rd” field.

For the “T3 ADD”, “T3 ADDS”, and “T4” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP.

<const>

For the “A1 ADD” and “A1 ADDS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T3 ADD” and “T3 ADDS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

<imm8>

Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the “imm8” field as <imm8>/4.

SP

Is the stack pointer.

<imm7>

Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the “imm7” field as <imm7>/4.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), imm32, '0');
    if d == 15 then                // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

C5.1.9 ADD, ADDS (SP plus register)

Add to SP (register) adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

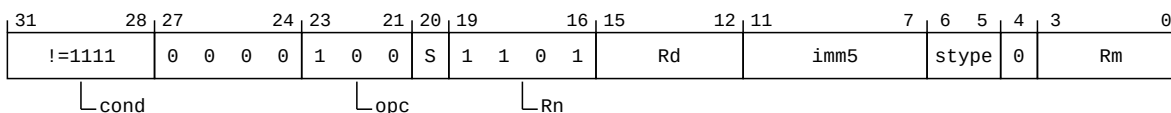
If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The ADD variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ADDS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from `SPSR_<current_mode>`.
 - The PE checks `SPSR_<current_mode>` for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, and T3).

A1



Encoding for the ADD, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

ADD {<c>} {<q>} {<Rd>}, SP, <Rm>, RRX

Encoding for the ADD, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

ADD{<c>}{<q>}{<Rd>,<Rm>,<shift>#<amount>}

Encoding for the ADDS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

ADDS {<c>} {<q>} {<Rd>}, SP, <Rm>, RRX

Encoding for the ADDS, shift or rotate by value variant

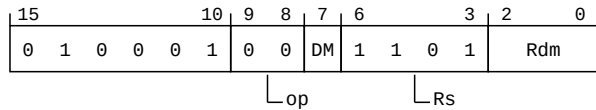
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

ADDS{<c>}{<q>}{<Rd>, }SP, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift t, shift n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1



Encoding

ADD{<c>}{<q>} {<Rdm>, }SP, <Rdm>

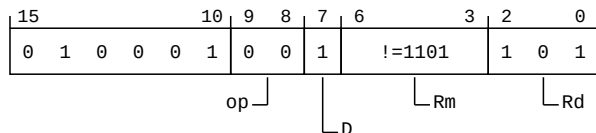
Decode for this encoding

```

let d : integer = UInt(DM::Rdm);
let m : integer = UInt(DM::Rdm);
let setflags : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if d == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;

```

T2



Encoding

ADD{<c>}{<q>} {SP, }SP, <Rm>

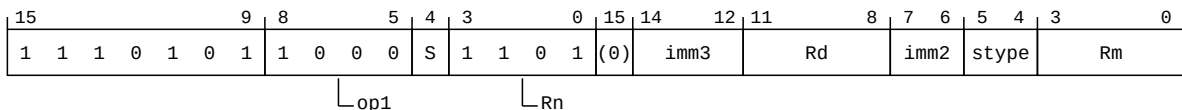
Decode for this encoding

```

if Rm == '1101' then See("encoding T1"); end;
let d : integer = 13;
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;

```

T3



Encoding for the ADD, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

ADD{<c>}{<q>} {<Rd>, }SP, <Rm>, RRX

Encoding for the ADD, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ADD{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

```
ADD{<c>}.W {<Rd>, }SP, <Rm> // (<Rd>, <Rm> can be represented in T1 or T2)
```

Encoding for the ADDS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

```
ADDS{<c>}{<q>} {<Rd>, }SP, <Rm>, RRX
```

Encoding for the ADDS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ADDS{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMN (register)"); end;
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, and “A1 ADDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ADDS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP.

<Rm>

For the “A1 ADD, rotate right with extend”, “A1 ADD, shift or rotate by value”, “A1 ADDS, rotate right with extend”, “A1 ADDS, shift or rotate by value”, and “T2” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 ADD, rotate right with extend”, “T3 ADD, shift or rotate by value”, “T3 ADDS, rotate right with extend”, and “T3 ADDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 ADD, shift or rotate by value” and “A1 ADDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T3 ADD, shift or rotate by value” and “T3 ADDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdm>

Is the general-purpose destination and second source register, encoded in the “Rdm” field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

SP

Is the stack pointer.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

C5.1.10 ADR

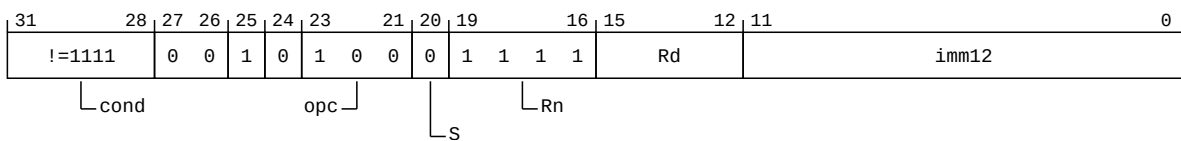
Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

This instruction is used by the alias [SUB \(immediate, from PC\)](#).

This instruction is used by the pseudo-instruction [ADD \(immediate, to PC\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



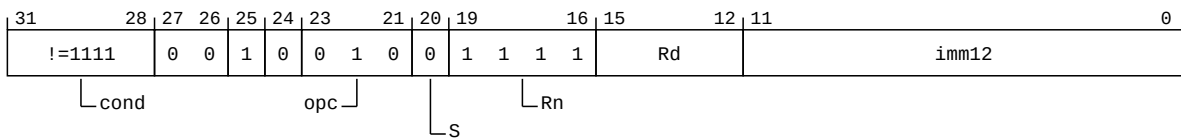
Encoding

ADR{<c>}{<q>} <Rd>, <label>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = A32ExpandImm(imm12);
let add : boolean = TRUE;
```

A2



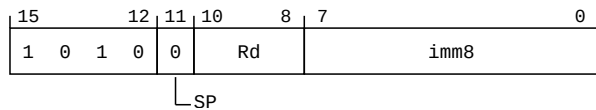
Encoding

ADR{<c>}{<q>} <Rd>, <label>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = A32ExpandImm(imm12);
let add : boolean = FALSE;
```

T1



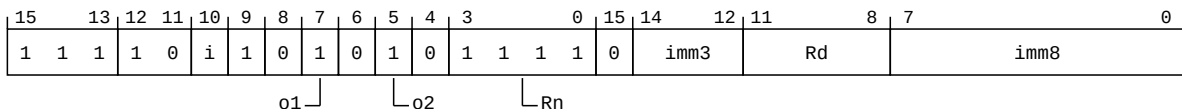
Encoding

ADR{<c>}{<q>} <Rd>, <label>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let add : boolean = TRUE;
```

T2



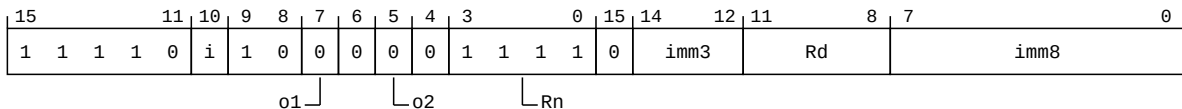
Encoding

ADR{<c>}{<q>} <Rd>, <label>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
let add : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

T3



Encoding

ADR{<c>}{<q>} <Rd>, <label>

ADR{<c>}.W <Rd>, <label> // (<Rd>, <label> can be presented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);  
let imm32 : bits(32) = ZeroExtend{(i::imm3::imm8)};  
let add : boolean = TRUE;  
// Armv8-A removes UNPREDICTABLE for R13  
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1” and “A2” variants: is the general-purpose destination register, encoded in the “Rd” field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T1”, “T2”, and “T3” variants: is the general-purpose destination register, encoded in the “Rd” field.

<label>

For the “A1” and “A2” variants: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the ADR instruction to this label.

If the offset is zero or positive, encoding A1 is used, with `imm32` equal to the offset.

If the offset is negative, encoding A2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are any of the constants described in [Modified immediate constants in A32 instructions](#).

For the “T1” variant: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For the “T2” and “T3” variants: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the ADR instruction to this label.

If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset.

If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`.

Permitted values of the size of the offset are 0–4095.

The instruction aliases permit the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Alias Conditions

Alias	Of variant	Is preferred when
ADD (immediate, to PC)		Never
SUB (immediate, from PC)	A2	imm12 == '000000000000'
SUB (immediate, from PC)	T2	i : imm3 : imm8 == '000000000000'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                             else (AlignDownSize(PC32(),4) - imm32));
    if d == 15 then          // Can only occur for A32 encodings
        ALUWritePC(result);
    else
        R(d) = result;
    end;
end;
```

C5.1.11 AND, ANDS (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

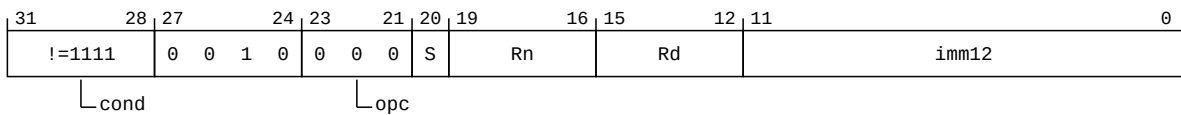
If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the AND variant

Applies when (S == 0)

AND{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the ANDS variant

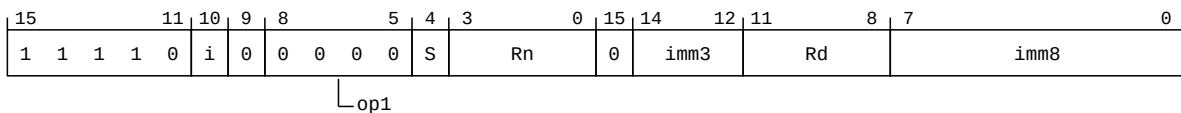
Applies when (S == 1)

ANDS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding for the AND variant

Applies when (S == 0)

AND{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the ANDS variant

Applies when (S == 1 && Rd != 1111)

ANDS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TST (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 AND” and “A1 ANDS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ANDS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the “T1 AND” and “T1 ANDS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 AND” and “A1 ANDS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 AND” and “T1 ANDS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 AND” and “A1 ANDS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1 AND” and “T1 ANDS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                             else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) AND imm32;
    if d == 15 then                // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

C5.1.12 AND, ANDS (register)

Bitwise AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

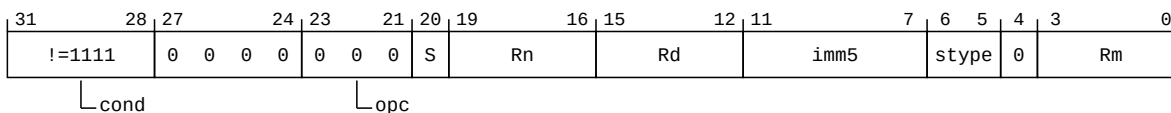
If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The AND variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ANDS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the AND, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

AND {<c>} {<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the AND, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

AND{<c>}{<q>}{<Rd>,<Rn>,<Rm>,<shift>#<amount>}

Encoding for the ANDS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

ANDS {<c>} {<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ANDS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

ANDS{<c>}{<q>}{<Rd>,<Rn>,<Rm>{,<shift>#<amount>}

Decode for all variants of this encoding

```

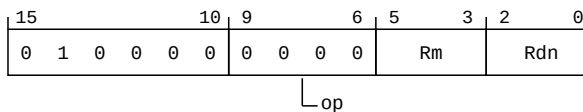
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

let setflags : boolean = (S == '1');

let (shift t, shift n) : (SRTyp, integer) = DecodeImmShift(stype, imm5);

```

T1



Encoding

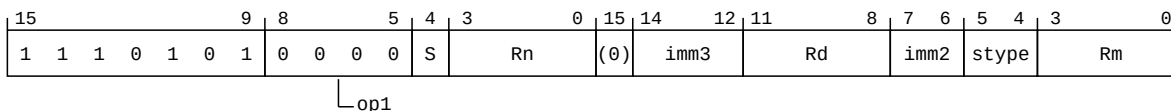
AND<c>{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())

ANDS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the AND, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

AND{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the AND, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

AND{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

AND<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the ANDS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

ANDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ANDS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ANDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

ANDS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TST (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 AND, rotate right with extend”, “A1 AND, shift or rotate by value”, “A1 ANDS, rotate right with extend”, and “A1 ANDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ANDS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T2 AND, rotate right with extend”, “T2 AND, shift or rotate by value”, “T2 ANDS, rotate right with extend”, and “T2 ANDS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 AND, rotate right with extend”, “A1 AND, shift or rotate by value”, “A1 ANDS, rotate right with extend”, and “A1 ANDS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 AND, rotate right with extend”, “T2 AND, shift or rotate by value”, “T2 ANDS, rotate right with extend”, and “T2 ANDS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 AND, rotate right with extend”, “A1 AND, shift or rotate by value”, “A1 ANDS, rotate right with extend”, and “A1 ANDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 AND, rotate right with extend”, “T2 AND, shift or rotate by value”, “T2 ANDS, rotate right with extend”, and “T2 ANDS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

type	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 AND, shift or rotate by value” and “A1 ANDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 AND, shift or rotate by value” and “T2 ANDS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

In T32 assembly:

- Outside an IT block, if ANDS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ANDS <Rd>, <Rn> had been written.
- Inside an IT block, if AND<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though AND<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .w qualifier.

Operation

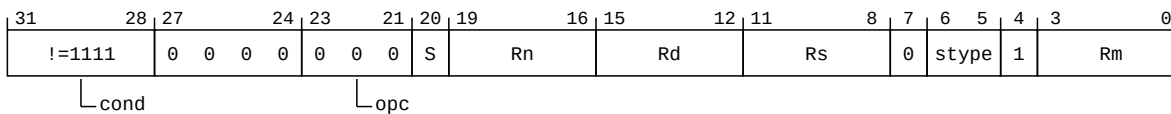
```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;

```

C5.1.13 AND, ANDS (register-shifted register)

Bitwise AND (register-shifted register) performs a bitwise AND of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

ANDS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

AND{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

C5.1.14 ASR (immediate)

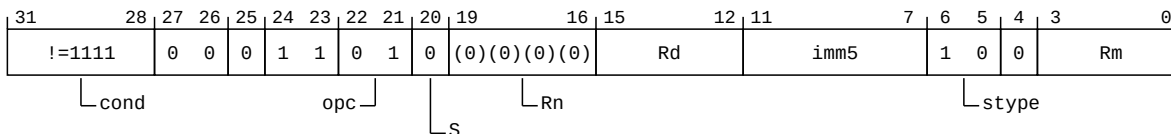
Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1



Encoding for the MOV, shift or rotate by value variant

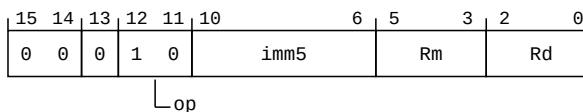
ASR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

T2



Encoding for the T2 variant

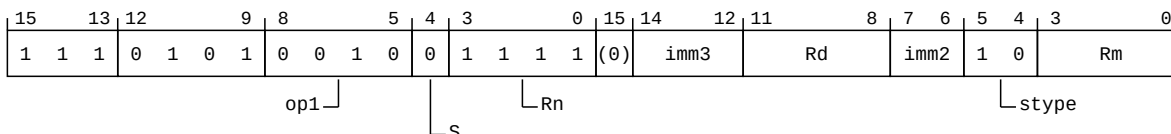
ASR<c>{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV<c>{<q>} <Rd>, <Rm>, ASR #<imm>

and is the preferred disassembly when `InITBlock()`.

T3



Encoding for the MOV, shift or rotate by value variant

```
ASR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>
```

```
ASR<c>.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOV, shift or rotate by value” and “T2” variants: is the shift amount, in the range 1 to 32, encoded in the “imm5” field as <imm> modulo 32.

For the “T3 MOV, shift or rotate by value” variant: is the shift amount, in the range 1 to 32, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.15 ASR (register)

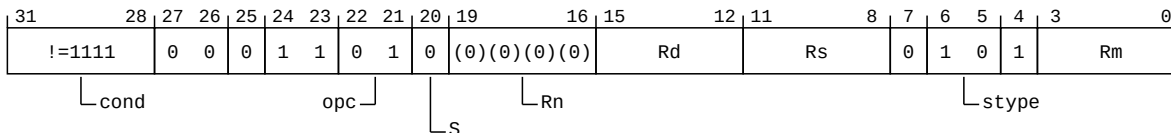
Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Not flag setting variant

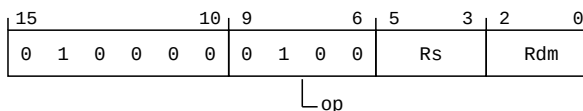
ASR{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

T1



Encoding for the Arithmetic shift right variant

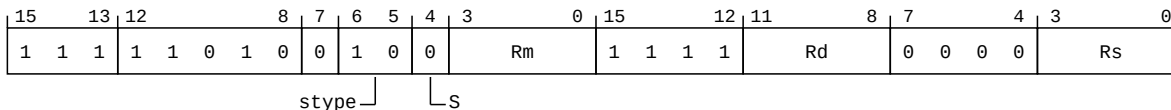
ASR<c>{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs>

and is the preferred disassembly when `InITBlock()`.

T2



Encoding for the Not flag setting variant

```
ASR{<c>}{<q>}{<Rd>, }<Rm>, <Rs>
```

```
ASR<c>.W {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOV{<c>}{<q>}{<Rd>, <Rm>, ASR <Rs>}
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

C5.1.16 ASRS (immediate)

Arithmetic Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

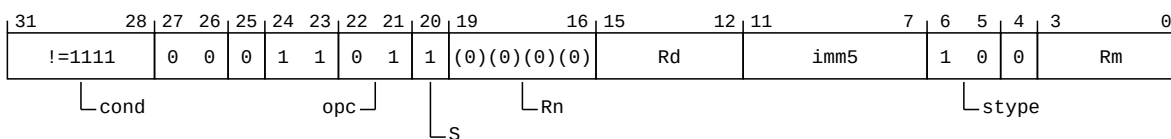
- The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1



Encoding for the MOVs, shift or rotate by value variant

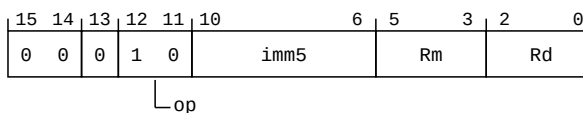
ASRS{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>

and is always the preferred disassembly.

T2



Encoding for the T2 variant

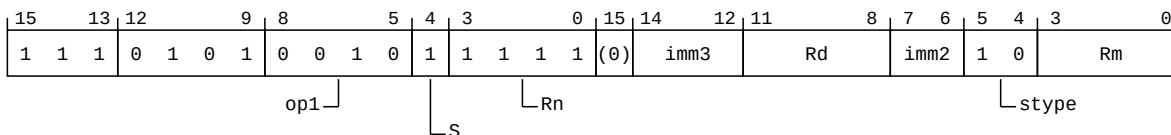
ASRS{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS<c>{<q>} <Rd>, <Rm>, ASR #<imm>

and is the preferred disassembly when !InITBlock().

T3



Encoding for the MOVs, shift or rotate by value variant

```
ASRS{<c>}{<q>} {<Rd>, }<Rm>, #<imm>
```

```
ASRS.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOVs, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the “T2” and “T3 MOVs, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVs, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOVs, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOVs, shift or rotate by value” and “T2” variants: is the shift amount, in the range 1 to 32, encoded in the “imm5” field as <imm> modulo 32.

For the “T3 MOVs, shift or rotate by value” variant: is the shift amount, in the range 1 to 32, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of [MOV](#), [MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.17 ASRS (register)

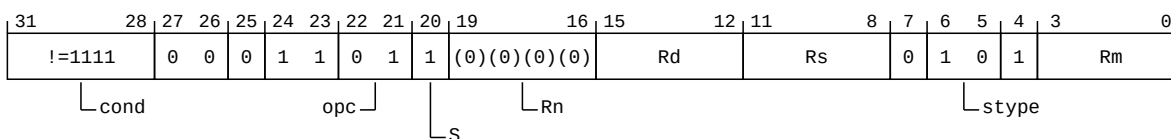
Arithmetic Shift Right, setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Flag setting variant

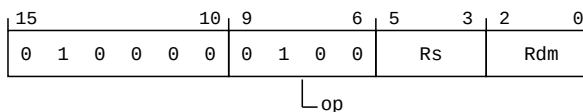
ASRS{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>

and is always the preferred disassembly.

T1



Encoding for the Arithmetic shift right variant

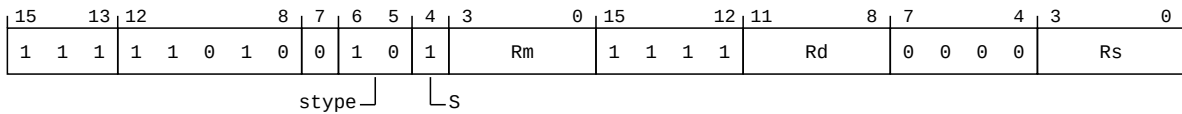
ASRS{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>

and is the preferred disassembly when !InITBlock().

T2



Encoding for the Flag setting variant

ASRS{<c>}{<q>}{<Rd>,<Rm>,<Rs>}

```
ASRS.W {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

 $\langle R_s \rangle$

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

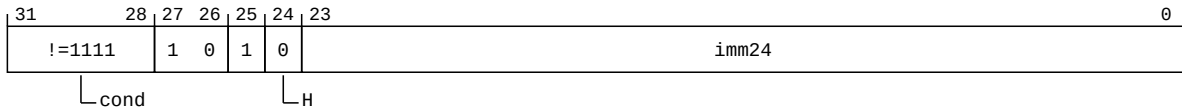
The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

C5.1.18 B

Branch causes a branch to a target address.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1, T2, T3, and T4).

A1



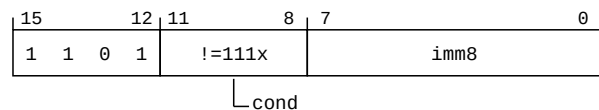
Encoding

$$B\{\langle c \rangle\} \{\langle q \rangle\} \quad \langle \text{label} \rangle$$

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{}(imm24::'00');
```

T1



Encoding

B<c>{<g>} <label> // (Not permitted in IT block)

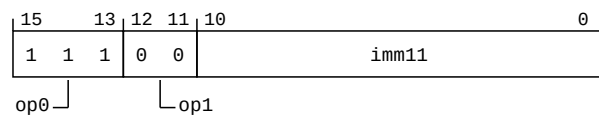
Decode for this encoding

```

if cond == 'l110' then See("UDF"); end;
if cond == 'l111' then See("SVC"); end;
let imm32 : bits(32) = SignExtend{(imm8::'0')};
if InITBlock() then UnpredictableProcedure(); end;

```

T2



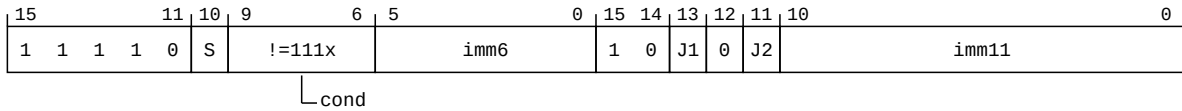
Encoding

```
B{<c>}{<q>} <label> // (Outside or last in IT block)
```

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{{imm11::'0'}};
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T3



Encoding

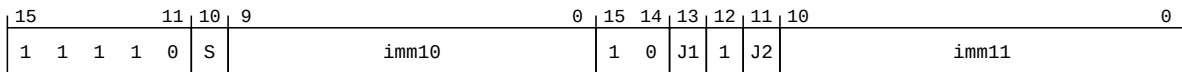
B<c>.W <label> // (Not permitted in IT block, and <label> can be represented in T1)

B<c>{<q>} <label> // (Not permitted in IT block)

Decode for this encoding

```
if cond[3:1] == '111' then See("Related encodings"); end;
let imm32 : bits(32) = SignExtend{(S::J2::J1::imm6::imm11::'0')};
if InITBlock() then UnpredictableProcedure(); end;
```

T4



Encoding

$$B\{\langle c \rangle\} \{\langle q \rangle\} \quad \langle \text{label} \rangle$$

```
B{<c>}.W <label> // (<label> can be represented in T2)
```

Decode for this encoding

```

let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{(S::I1::I2::imm10::imm11::'0')};
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Branches and miscellaneous control](#).

Assembler Symbols

<c>

For the “A1”, “T2”, and “T4” variants: see [Standard assembler syntax fields](#).

For the “T1” and “T3” variants: see [Standard assembler syntax fields](#). Must not be `AL` or omitted.

<q>

See [Standard assembler syntax fields](#).

<label>

For the “A1” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are multiples of 4 in the range -33554432 to 33554428 .

For the “T1” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range -256 to 254 .

For the “T2” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range -2048 to 2046 .

For the “T3” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are even numbers in the range -1048576 to 1048574 .

For the “T4” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset.

Permitted offsets are even numbers in the range -16777216 to 16777214 .

Operation

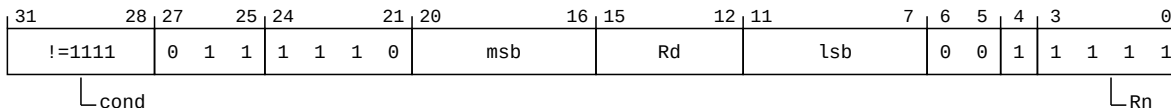
```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC32() + imm32, BranchType_DIR);
end;
```

C5.1.19 BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>

Decode for this encoding

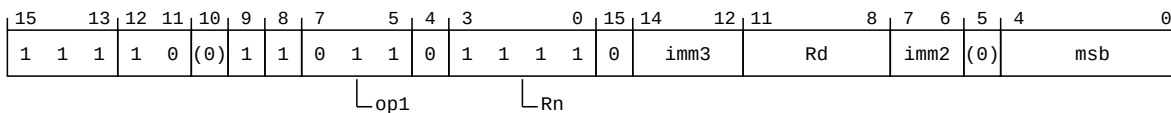
```
let d : integer = UInt(Rd);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(lsb);
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $msbit < lsbit$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding

BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<lsb>

For the “A1” variant: is the least significant bit to be cleared, in the range 0 to 31, encoded in the “lsb” field.

For the “T1” variant: is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the “imm3:imm2” field.

<width>

Is the number of bits to be cleared, in the range 1 to 32-`<lsb>`, encoded in the “msb” field as `<lsb>+<width>-1`.

Operation

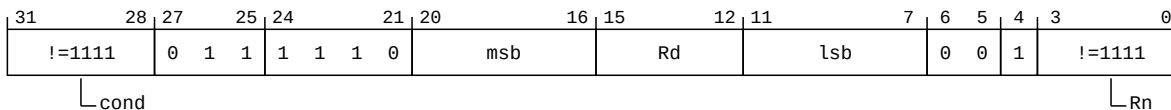
```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[msbit:lsbit] = Replicate{(msbit-lsbit)+1}('0');
    // Other bits of R[d] are unchanged
end;
```

C5.1.20 BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```

if Rn == '1111' then See("BFC"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(lsb);
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;

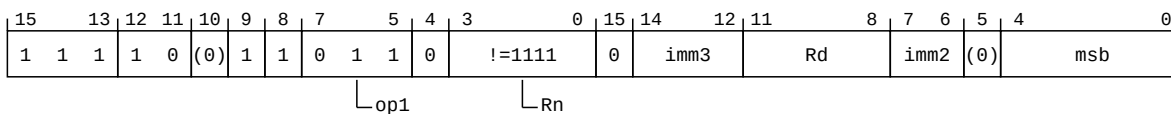
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```

if Rn == '1111' then See("BFC"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let msbit : integer{} = UInt(msb);
let lsbit : integer{} = UInt(imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
if msbit < lsbit then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “A1” variant: is the least significant destination bit, in the range 0 to 31, encoded in the “lsb” field.

For the “T1” variant: is the least significant destination bit, in the range 0 to 31, encoded in the “imm3:imm2” field.

<width>

Is the number of bits to be copied, in the range 1 to 32-`<lsb>`, encoded in the “msb” field as `<lsb>+<width>-1`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[msbit:lsbit] = R(n)[(msbit-lsbit):0];
    // Other bits of R[d] are unchanged
end;
```

C5.1.21 BIC, BICS (immediate)

Bitwise Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register.

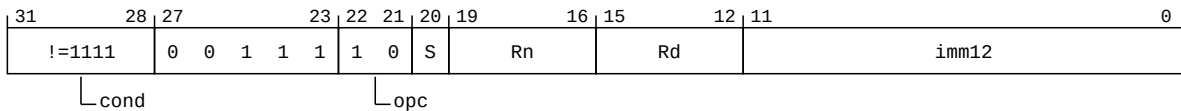
If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the BIC variant

Applies when (S == 0)

BIC{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the BICS variant

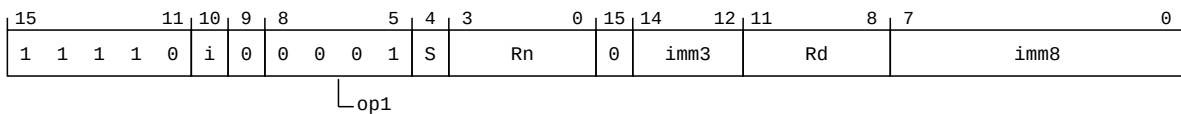
Applies when (S == 1)

BICS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding for the BIC variant

Applies when (S == 0)

BIC{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the BICS variant

Applies when (S == 1)

BICS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 BIC” and “A1 BICS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the BICS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T1 BIC” and “T1 BICS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 BIC” and “A1 BICS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 BIC” and “T1 BICS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 BIC” and “A1 BICS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1 BIC” and “T1 BICS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                              else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) AND NOT(imm32);
    if d == 15 then                // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

C5.1.22 BIC, BICS (register)

Bitwise Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

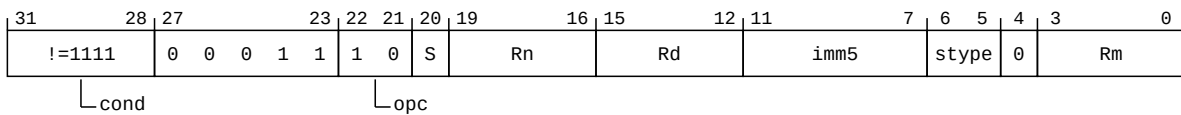
If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The BIC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The BICS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the BIC, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the BIC, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the BICS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the BICS, shift or rotate by value variant

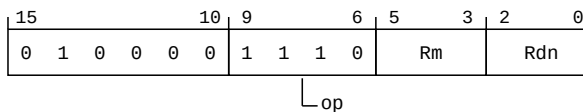
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1



Encoding

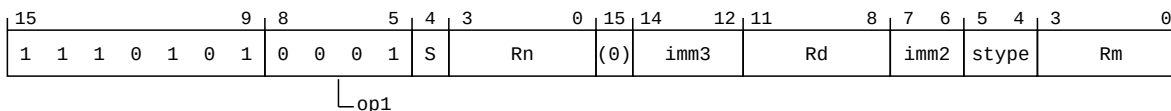
BIC<c>{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())

BICS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the BIC, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the BIC, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

BIC<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the BICS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the BICS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

BICS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 BIC, rotate right with extend”, “A1 BIC, shift or rotate by value”, “A1 BICS, rotate right with extend”, and “A1 BICS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the BICS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T2 BIC, rotate right with extend”, “T2 BIC, shift or rotate by value”, “T2 BICS, rotate right with extend”, and “T2 BICS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 BIC, rotate right with extend”, “A1 BIC, shift or rotate by value”, “A1 BICS, rotate right with extend”, and “A1 BICS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 BIC, rotate right with extend”, “T2 BIC, shift or rotate by value”, “T2 BICS, rotate right with extend”, and “T2 BICS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 BIC, rotate right with extend”, “A1 BIC, shift or rotate by value”, “A1 BICS, rotate right with extend”, and “A1 BICS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 BIC, rotate right with extend”, “T2 BIC, shift or rotate by value”, “T2 BICS, rotate right with extend”, and “T2 BICS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 BIC, shift or rotate by value” and “A1 BICS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T2 BIC, shift or rotate by value” and “T2 BICS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

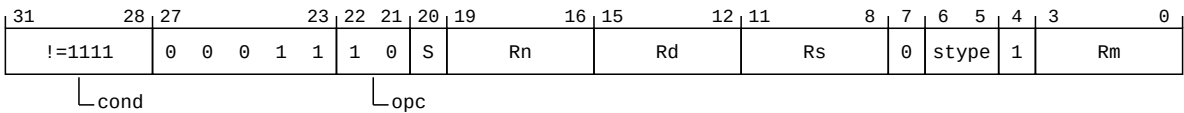
Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND NOT(shifted);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;
```

C5.1.23 BIC, BICS (register-shifted register)

Bitwise Bit Clear (register-shifted register) performs a bitwise AND of a register value and the complement of a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

BICS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

BIC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SType = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND NOT(shifted);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

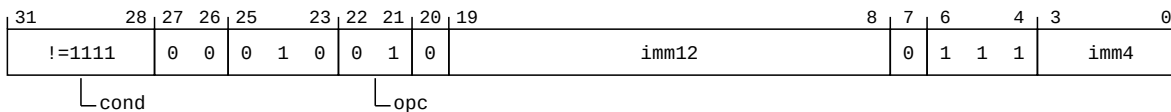
C5.1.24 BKPT

Breakpoint causes a Breakpoint Instruction exception.

Breakpoint is always unconditional, even when inside an IT block.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

BKPT{<q>} {#}<imm>

Decode for this encoding

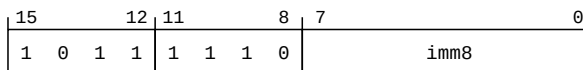
```
let imm16 : bits(16) = imm12::imm4;
if cond != '1110' then UnpredictableProcedure(); end; // BKPT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1



Encoding

BKPT{<q>} {#}<imm>

Decode for this encoding

```
let imm16 : bits(16) = ZeroExtend{}(imm8);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#). A BKPT instruction must be unconditional.

<imm>

For the “A1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm12:imm4” field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Is ignored otherwise.

For the “T1” variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field. This value:

- Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64.
- Is ignored otherwise.

Operation

```
EncodingSpecificOperations();  
AArch32\_SoftwareBreakpoint(imm16);
```

C5.1.25 BL, BLX (immediate)

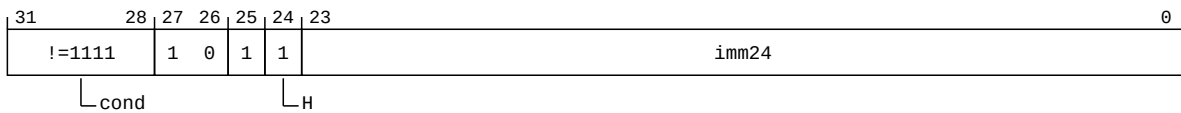
Branch with Link and optional Exchange (immediate)

Branch with Link calls a subroutine at a PC-relative address, and setting LR to the return address.

Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, setting LR to the return address, and changes the instruction set from A32 to T32, or from T32 to A32.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding

```
BL{<c>} {<q>}    <label>
```

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{}(imm24::'00');
let targetInstrSet : InstrSet = InstrSet_A32;
```

A2



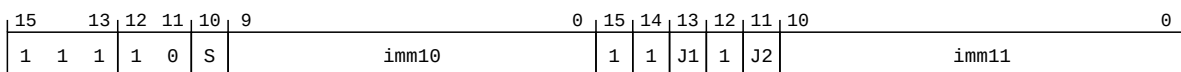
Encoding

```
BLX{<c>}{<q>}  <label>
```

Decode for this encoding

```
let imm32 : bits(32) = SignExtend{}(imm24::H::'0');
let targetInstrSet : InstrSet = InstrSet T32;
```

T1



Encoding

$$\text{BL}\{\langle c \rangle\} \{\langle q \rangle\} \quad \langle \text{label} \rangle$$

Decode for this encoding

```
let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{(S::I1::I2::imm10::imm11::'0')};
let targetInstrSet : InstrSet = InstrSet_T32;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2

15	13	12	11	10	9											0	15	14	13	12	11	10											1	0
1	1	1	1	0	S	imm10H										1	1	J1	0	J2	imm10L										H			

Encoding

```
BLX{<c>}{<q>} <label>
```

Decode for this encoding

```
if H == '1' then Undefined(); end;
let I1 : bit = NOT(J1 XOR S);
let I2 : bit = NOT(J2 XOR S);
let imm32 : bits(32) = SignExtend{(S::I1::I2::imm10H::imm10L::'00')};
let targetInstrSet : InstrSet = InstrSet_A32;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1”, “T1”, and “T2” variants: see [Standard assembler syntax fields](#).
For the “A2” variant: see [Standard assembler syntax fields](#). <c> must be AL or omitted.

<q>

See [Standard assembler syntax fields](#).

<label>

For the “A1” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding that sets imm32 to that offset.

Permitted offsets are multiples of 4 in the range –33554432 to 33554428.

For the “A2” variant: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset.

Permitted offsets are even numbers in the range –33554432 to 33554430.

For the “T1” variant: the label of the instruction that is to be branched to.

The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding with imm32 set to that offset.

Permitted offsets are even numbers in the range –16777216 to 16777214.

For the “T2” variant: the label of the instruction that is to be branched to.

The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the BLX instruction to this label, then selects an encoding with `imm32` set to that offset.

Permitted offsets are multiples of 4 in the range -16777216 to 16777212 .

Operation

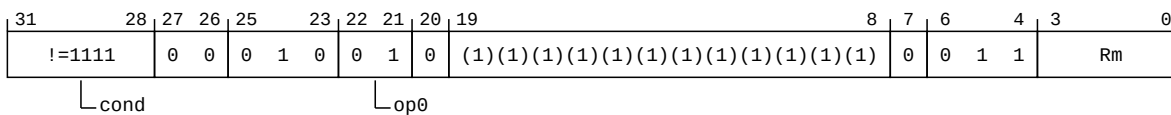
```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_A32 then
        LR() = PC32() - 4;
    else
        LR() = PC32()[31:1] :: '1';
    end;
    var targetAddress : bits(32);
    if targetInstrSet == InstrSet_A32 then
        targetAddress = AlignDownSize(PC32(), 4) + imm32;
    else
        targetAddress = PC32() + imm32;
    end;
    SelectInstrSet(targetInstrSet);
    BranchWritePC(targetAddress, BranchType_DIRCALL);
end;
```

C5.1.26 BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address specified in the register, and if necessary changes to the instruction set indicated by bit[0] of the register value. If the value in bit[0] is 0, the instruction set after the branch will be A32. If the value in bit[0] is 1, the instruction set after the branch will be T32.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



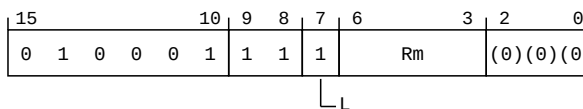
Encoding

$$\text{BLX}\{\langle c \rangle\} \{\langle q \rangle\} \quad \langle \text{Rm} \rangle$$

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

$$\text{BLX}\{\langle c \rangle\} \{\langle q \rangle\} \quad \langle \text{Rm} \rangle$$

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

 $\langle R_m \rangle$

Is the general-purpose register holding the address to be branched to, encoded in the “Rm” field.

Operation

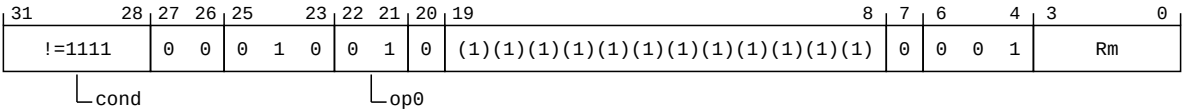
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let target : bits(32) = R(m);
    var next_instr_addr : bits(32);
    if CurrentInstrSet() == InstrSet_A32 then
        next_instr_addr = PC32() - 4;
        LR() = next_instr_addr;
    else
        next_instr_addr = PC32() - 2;
        LR() = next_instr_addr[31:1] :: '1';
    end;
    BXWritePC(target, BranchType_INDCALL);
end;
```

C5.1.27 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



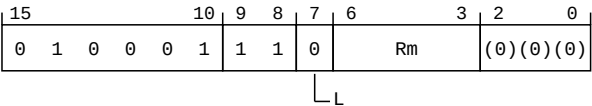
Encoding

BX{<c>}{<q>}<Rm>

Decode for this encoding

let m : integer = UInt(Rm);

T1



Encoding

BX{<c>}{<q>}<Rm>

Decode for this encoding

let m : integer = UInt(Rm);
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rm>

For the “A1” variant: is the general-purpose register holding the address to be branched to, encoded in the “Rm” field. The PC can be used.

For the “T1” variant: is the general-purpose register holding the address to be branched to, encoded in the “Rm” field. The PC can be used.

Note

If <Rm> is the PC at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the `BXWritePC()` pseudocode function has bits<1:0> = '10'.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R(m), BranchType_INDIR);
end;
```

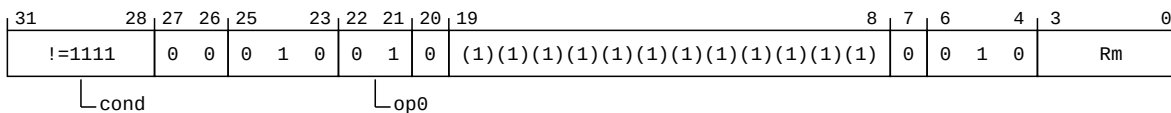
C5.1.28 BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.

BXJ behaves as a **BX** instruction, see **BX**. This means it causes a branch to an address and instruction set specified by a register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



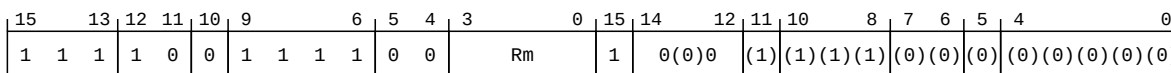
Encoding

$$\text{BXJ}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle \text{Rm} \rangle$$

Decode for this encoding

```
let m : integer = UInt(Rm);
if m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

$$\text{BXJ}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle \text{Rm} \rangle$$

Decode for this encoding

```
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the `CONSTRAINED UNPREDICTABLE` behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rm>

Is the general-purpose register holding the address to be branched to, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R(m), BranchType_INDIR);
end;
```

C5.1.29 CBNZ, CBZ

Compare and Branch on Nonzero or Zero

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

15	12	11	10	9	8	7	3	2	0
1	0	1	1	op	0	i	1	imm5	Rn

Encoding for the CBNZ variant

Applies when (op == 1)

CBNZ{<q>} <Rn>, <label>

Encoding for the CBZ variant

Applies when (op == 0)

CBZ{<q>} <Rn>, <label>

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(i::imm5::'0')};
let nonzero : boolean = (op == '1');
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose register to be tested, encoded in the “Rn” field.

<label>

Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 and in the range 0 to 126, is encoded as “i:imm5” times 2.

Operation

```
EncodingSpecificOperations();
if nonzero != IsZero(R(n)) then
    CBWritePC(PC32() + imm32);
end;
```

C5.1.30 CLREX

Clear-Exclusive clears the local monitor of the executing PE.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	21	20	19	8	7	4	3	0	
1	1	1	1	0	1	0	1	(1)(1)(1)(1)(1)(1)(1)(1)(0)(0)(0)(0)	0	0	0	1	(1)(1)(1)(1)
opcode										option			

Encoding

CLREX{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1

15	13	12	11	10	9	6	5	4	3	0	15	14	12	11	10	8	7	4	3	0		
1	1	1	1	0	0	1	1	1	0	1	1	(1)(1)(1)(1)	1	0	(0)0	(1)	(1)(1)(1)(1)	0	0	1	0	(1)(1)(1)(1)
opc																option						

Encoding

CLREX{<c>}{<q>}

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). Must be AL or omitted.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

Operation

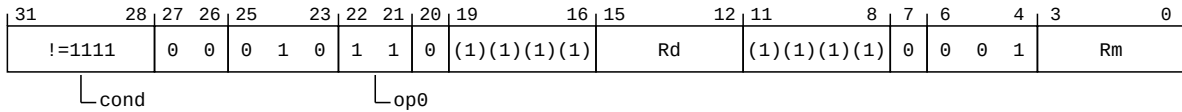
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
end;
```

C5.1.31 CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



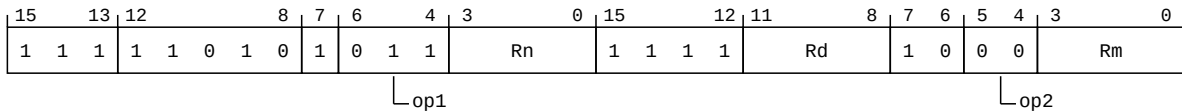
Encoding

CLZ{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

CLZ{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```

let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $m \neq n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes with the additional decode: $m = \text{UInt}(R_n)$.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” variant: is the general-purpose source register, encoded in the “Rm” field.

For the “T1” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

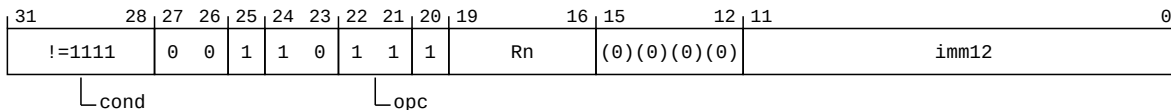
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = CountLeadingZeroBits(R(m));
    R(d) = result[31:0];
end;
```

C5.1.32 CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



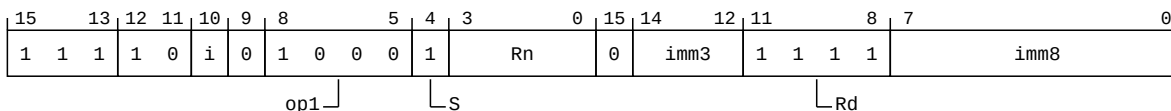
Encoding

CMN{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1



Encoding

CMN{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

For the “A1” variant: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1” variant: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1” variant: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

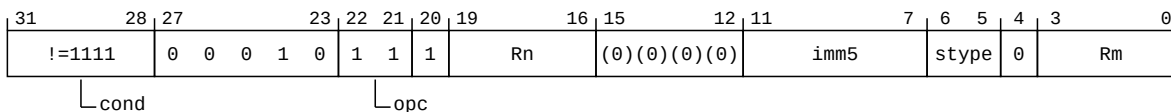
```
if ConditionPassed() then
    EncodingSpecificOperations();
let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), imm32, '0');
PSTATE.[N,Z,C,V] = nzcvc;
end;
```

C5.1.33 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

CMN{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

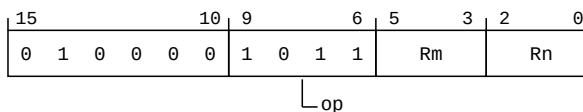
Applies when (!(imm5 == 00000 && stype == 11))

CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1



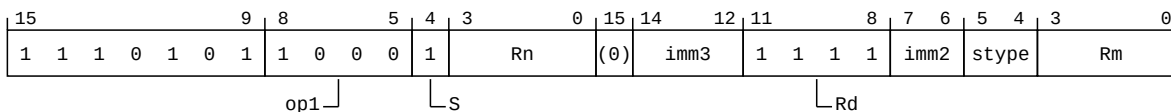
Encoding

CMN{<c>}{<q>} <Rn>, <Rm>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the Rotate right with extend variant

Applies when (imm3 == 000 && imm2 == 00 && stype == 11)

CMN { <c> } { <q> } <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00 && stype == 11))`

CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

CMN{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```

let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the `CONSTRAINED UNPREDICTABLE` behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

 $\langle R_n \rangle$

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 Rotate right with extend”, and “T2 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 Rotate right with extend”, and “T2 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    PSTATE.[N,Z,C,V] = nzcvc;
end;
```


Operation

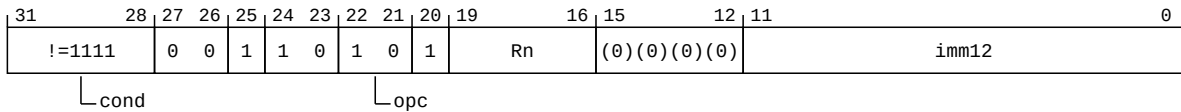
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), shifted, '0');
    PSTATE.[N,Z,C,V] = nzcv;
end;
```

C5.1.35 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



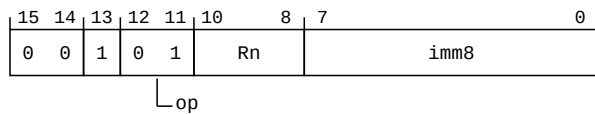
Encoding

CMP{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1



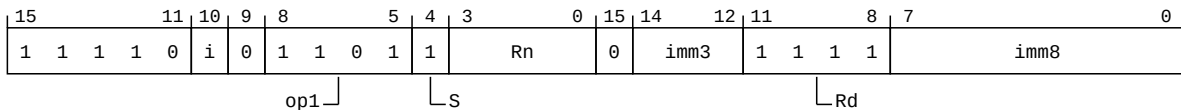
Encoding

CMP{<c>}{<q>} <Rn>, #<imm8>

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
```

T2



Encoding

CMP{<c>}{<q>} <Rn>, #<const>

CMP{<c>}.W <Rn>, #<const> // (<Rd>, <const> can be represented in T1)

Decode for this encoding

```
let n : integer = UInt(Rn);  
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);  
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

For the “A1” variant: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1” variant: is a general-purpose source register, encoded in the “Rn” field.

For the “T2” variant: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1” variant: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T2” variant: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

<imm8>

Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

Operation

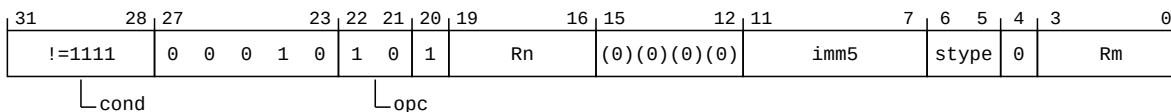
```
if ConditionPassed() then  
    EncodingSpecificOperations();  
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), '1');  
    PSTATE.[N,Z,C,V] = nzcvc;  
end;
```

C5.1.36 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

CMP{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

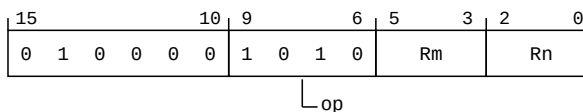
Applies when (!(imm5 == 00000 && stype == 11))

CMP{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1



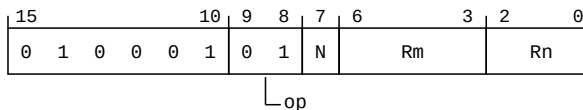
Encoding

CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> both from R0-R7)

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> not both from R0-R7)

Decode for this encoding

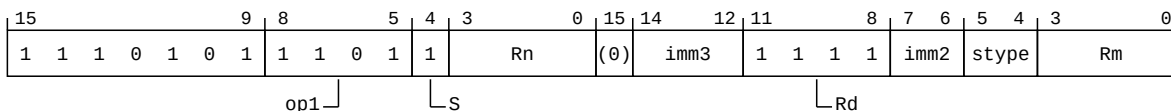
```
let n : integer = UInt(N::Rn);
let m : integer = UInt(Rm);
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if n < 8 && m < 8 then UnpredictableProcedure(); end;
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $n < 8$ && $m < 8$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The condition flags become UNKNOWN.

T3



Encoding for the Rotate right with extend variant

Applies when $(imm3 == 000 \ \&\& \ imm2 == 00 \ \&\& \ stype == 11)$

CMP{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

Applies when $(!(imm3 == 000 \ \&\& \ imm2 == 00 \ \&\& \ stype == 11))$

CMP{<c>}{<q>} <Rn>, <Rm>, <shift> #<amount>

CMP{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1 or T2)

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1”, “T3 Rotate right with extend”, and “T3 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

For the “T2” variant: is the first general-purpose source register, encoded in the “N:Rn” field.

<Rm>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2”, “T3 Rotate right with extend”, and “T3 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

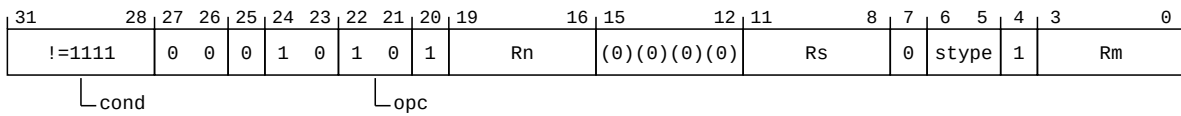
For the “T3 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    PSTATE.[N,Z,C,V] = nzcvc;
end;
```

C5.1.37 CMP (register-shifted register)

Compare (register-shifted register) subtracts a register-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.



Encoding

CMP{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

```

let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SRTType = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

 $\langle R_n \rangle$

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<type>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

 $\langle R_s \rangle$

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcv) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    PSTATE.[N,Z,C,V] = nzcv;
end;
```

C5.1.38 CPS, CPSID, CPSIE

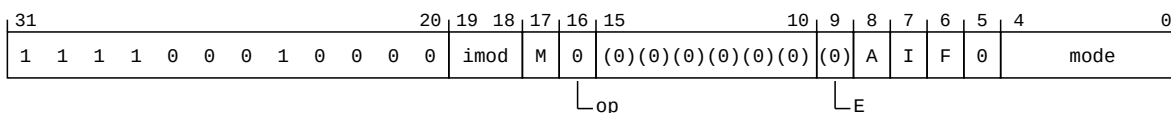
Change PE State changes one or more of the **PSTATE**.{A, I, F} interrupt mask bits and, optionally, the **PSTATE**.M mode field, without changing any other **PSTATE** bits.

CPS is treated as NOP if executed in User mode unless it is defined as being CONstrained UNPREDICTABLE elsewhere in this section.

The PE checks whether the value being written to PSTATE.M is legal. See [Illegal changes to PSTATE.M](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Change mode variant

Applies when (imod == 00 && M == 1)

```
CPS{<q>} #<mode> // (Cannot be conditional)
```

Encoding for the Interrupt disable variant

Applies when (imod == 11 && M == 0)

```
CPSID{<q>} <iflags> // (Cannot be conditional)
```

Encoding for the Interrupt disable and change mode variant

Applies when (imod == 11 && M == 1)

```
CPSID{<q>}    <iflags> , #<mode> // (Cannot be conditional)
```

Encoding for the Interrupt enable variant

Applies when (imod == 10 && M == 0)

```
CPSIE{<q>} <iflags> // (Cannot be conditional)
```

Encoding for the Interrupt enable and change mode variant

Applies when (imod == 10 && M == 1)

```
CPSIE{<q>}    <iflags> , #<mode> // (Cannot be conditional)
```

Decode for all variants of this encoding

```

if mode != '00000' && M == '0' then UnpredictableProcedure(); end;
if (imod[1] == '1' && A::I::F == '000') || (imod[1] == '0' && A::I::F != '000') then
    UnpredictableProcedure();
end;

let enable : boolean = (imod == '10');
let disable : boolean = (imod == '11');
let changemode : boolean = (M == '1');
let pemode : bits(5) = mode;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');

if (imod == '00' && M == '0') || imod == '01' then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `imod == '00' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `changemode = TRUE`.
- The instruction executes as described, and the value specified by `mode` is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '0'`.
- The instruction behaves as if `A:I:F` has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '1'`.
- The instruction behaves as if `A:I:F == '000'`.

T1

15	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0
0	0	1	1	im	(0)	A	I
						F	

└─ op

Encoding for the Interrupt disable variant

Applies when `(im == 1)`

`CPSID{<q>} <iflags> // (Not permitted in IT block)`

Encoding for the Interrupt enable variant

Applies when `(im == 0)`

`CPSIE{<q>} <iflags> // (Not permitted in IT block)`

Decode for all variants of this encoding

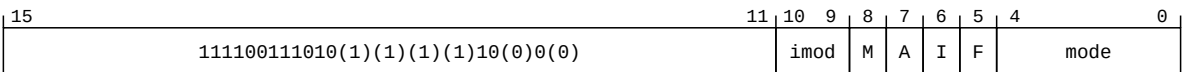
```
if A::I::F == '000' then UnpredictableProcedure(); end;
let enable : boolean = (im == '0');
let disable : boolean = (im == '1');
let changemode : boolean = FALSE;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');
let pemode : bits(5) = ARBITRARY : bits(5);
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If A:I:F == '000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

T2



Encoding for the Change mode variant

Applies when (imod == 00 && M == 1)

```
CPSID{<q>} #<mode> // (Not permitted in IT block)
```

Encoding for the Interrupt disable variant

Applies when (imod == 11 && M == 0)

```
CPSID.W <iflags> // (Not permitted in IT block)
```

Encoding for the Interrupt disable and change mode variant

Applies when (imod == 11 && M == 1)

```
CPSID{<q>} <iflags>, #<mode> // (Not permitted in IT block)
```

Encoding for the Interrupt enable variant

Applies when (imod == 10 && M == 0)

```
CPSIE.W <iflags> // (Not permitted in IT block)
```

Encoding for the Interrupt enable and change mode variant

Applies when (imod == 10 && M == 1)

```
CPSIE{<q>} <iflags>, #<mode> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```

if imod == '00' && M == '0' then See("Hint instructions"); end;
if mode != '00000' && M == '0' then UnpredictableProcedure(); end;
if (imod[1] == '1' && A::I::F == '000') || (imod[1] == '0' && A::I::F != '000') then
    UnpredictableProcedure();
end;
let enable : boolean = (imod == '10');
let disable : boolean = (imod == '11');
let changemode : boolean = (M == '1');
let pemode : bits(5) = mode;
let affectA : boolean = (A == '1');
let affectI : boolean = (I == '1');
let affectF : boolean = (F == '1');
if imod == '01' || InITBlock() then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `changemode = TRUE`.
- The instruction executes as described, and the value specified by `mode` is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '0'`.
- The instruction behaves as if `A:I:F` has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '1'`.
- The instruction behaves as if `A:I:F == '000'`.

Hint instructions: In encoding T2, if the `imod` field is `0b00` and the `M` bit is `0b0`, a hint instruction is encoded. To determine which hint instruction, see [Branches and miscellaneous control](#).

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<mode>

Is the number of the mode to change to, in the range 0 to 31, encoded in the “mode” field.

<iflags>

Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

- a** Sets the A bit in the instruction, causing the specified effect on [PSTATE.A](#), the SError interrupt mask bit.
- i** Sets the I bit in the instruction, causing the specified effect on [PSTATE.I](#), the IRQ interrupt mask bit.
- f** Sets the F bit in the instruction, causing the specified effect on [PSTATE.F](#), the FIQ interrupt mask bit.

Operation

```

if CurrentInstrSet() == InstrSet_A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != ELO then
        if enable then
            if affectA then PSTATE.A = '0'; end;
            if affectI then PSTATE.I = '0'; end;
            if affectF then PSTATE.F = '0'; end;
        end;
        if disable then
            if affectA then PSTATE.A = '1'; end;
            if affectI then PSTATE.I = '1'; end;
            if affectF then PSTATE.F = '1'; end;
        end;
        if changemode then
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32_WriteModeByInstr(pemode);
        end;
    end;
else
    EncodingSpecificOperations();
    if PSTATE.EL != ELO then
        if enable then
            if affectA then PSTATE.A = '0'; end;
            if affectI then PSTATE.I = '0'; end;
            if affectF then PSTATE.F = '0'; end;
        end;
        if disable then
            if affectA then PSTATE.A = '1'; end;
            if affectI then PSTATE.I = '1'; end;
            if affectF then PSTATE.F = '1'; end;
        end;
        if changemode then
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32_WriteModeByInstr(pemode);
        end;
    end;
end;
end;

```

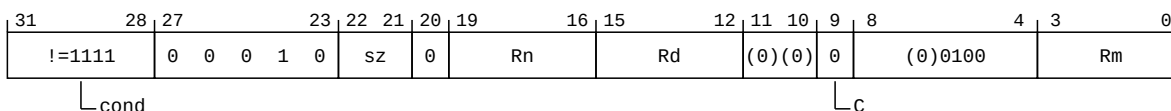
C5.1.39 CRC32

CRC32

CRC32 performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the CRC32B variant

Applies when (sz == 00)

CRC32B{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32H variant

Applies when (sz == 01)

CRC32H{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32W variant

Applies when (sz == 10)

CRC32W{ <q> } <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

```

if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
if cond != '1110' then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

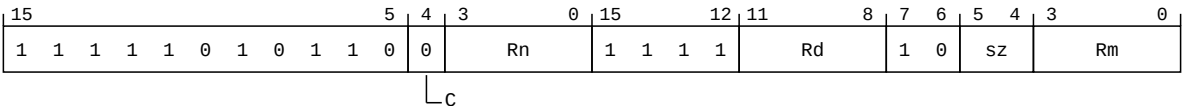
If `size == 64`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: `size = 32::`

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1



Encoding for the CRC32B variant

Applies when (sz == 00)

CRC32B{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32H variant

Applies when (sz == 01)

CRC32H{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32W variant

Applies when (sz == 10)

CRC32W{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#). A CRC32 instruction must be unconditional.

<Rd>

Is the general-purpose accumulator output register, encoded in the “Rd” field.

<Rn>

Is the general-purpose accumulator input register, encoded in the “Rn” field.

<Rm>

Is the general-purpose data source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    let acc : bits(32) = R(n);           // accumulator
    let val : bits(size) = R(m)[size-1:0]; // input value
    let poly : bits(32) = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)[31:0];
    let tempacc : bits(32+size) = BitReverse{32}(acc)::Zeros{size};
    let tempval : bits(32+size) = BitReverse{size}(val)::Zeros{32};
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R(d) = BitReverse{32}(Poly32Mod2{32+size}(tempacc XOR tempval, poly));
end;
```

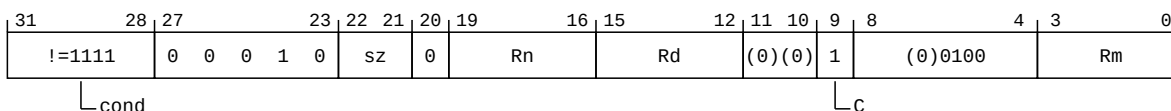
C5.1.40 CRC32C

CRC32C

CRC32C performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x1EDC6F41` is used for the CRC calculation.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the CRC32CB variant

Applies when (sz == 00)

CRC32CB{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32CH variant

Applies when (sz == 01)

CRC32CH{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32CW variant

Applies when (sz == 10)

CRC32CW{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

```

if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;
if cond != '1110' then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

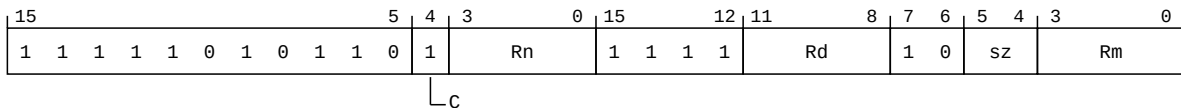
If `size == 64`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: `size = 32::`

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1



Encoding for the CRC32CB variant

Applies when (sz == 00)

CRC32CB{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32CH variant

Applies when (sz == 01)

CRC32CH{<q>} <Rd>, <Rn>, <Rm>

Encoding for the CRC32CW variant

Applies when (sz == 10)

CRC32CW{<q>} <Rd>, <Rn>, <Rm>

Decode for all variants of this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_CRC32) then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let size : integer{} = 8 << UInt(sz);
let crc32c : boolean = (C == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if size == 64 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == 64`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: `size = 32;`

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#). A CRC32C instruction must be unconditional.

<Rd>

Is the general-purpose accumulator output register, encoded in the “Rd” field.

<Rn>

Is the general-purpose accumulator input register, encoded in the “Rn” field.

<Rm>

Is the general-purpose data source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    let acc : bits(32) = R(n);           // accumulator
    let val : bits(size) = R(m)[size-1:0]; // input value
    let poly : bits(32) = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)[31:0];
    let tempacc : bits(32+size) = BitReverse{32}(acc)::Zeros{size};
    let tempval : bits(32+size) = BitReverse{size}(val)::Zeros{32};
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R(d) = BitReverse{32}(Poly32Mod2{32+size}(tempacc XOR tempval, poly));
end;
```

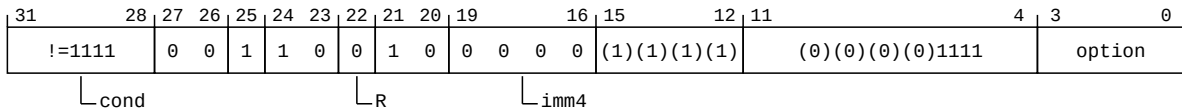
C5.1.41 **DBG**

Debug hint

DBG executes as a NOP. Arm deprecates any use of the DBG instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



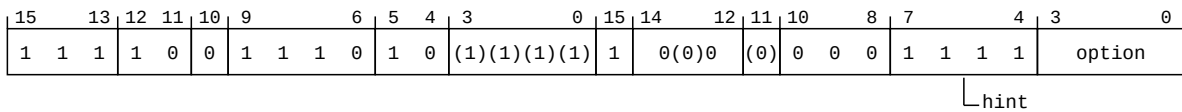
Encoding

```
DBG{<c>}{<q>} #<option>
```

Decode for this encoding

```
// DBG executes as a NOP. The 'option' field is ignored
```

T1



Encoding

```
DBG{<c>}{<q>} #<option>
```

Decode for this encoding

```
// DBG executes as a NOP. The 'option' field is ignored
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<option>

Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “option” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
end;
```

C5.1.42 DCPS1

Debug Change PE State to EL1 allows the debugger to move the PE into EL1 from EL0 or to a specific mode at the current Exception level.

DCPS1 is UNDEFINED if any of:

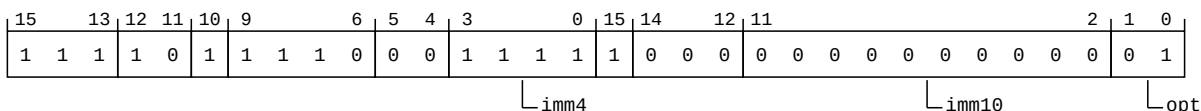
- The PE is in Non-debug state.
- EL2 is implemented, EL2 is implemented and enabled in the current Security state, and any of:
 - EL2 is using AArch32 and HCR.TGE is set to 1.
 - EL2 is using AArch64 and HCR_EL2.TGE is set to 1.

When the PE executes DCPS1 at EL0, EL1 or EL3:

- If EL3 or EL1 is using AArch32, the PE enters SVC mode and LR_svc, SPSR_svc, DLR, and DSPSR become UNKNOWN. If DCPS1 is executed in Monitor mode, SCR_NS is cleared to 0.
- If EL1 is using AArch64, the PE enters EL1 using AArch64, selects SP_EL1, and ELR_EL1, ESR_EL1, SPSR_EL1, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

When the PE executes `DCPS1` at EL2 the PE does not change mode, and `ELR_hyp`, `HSR`, `SPSR_hyp`, `DLR` and `DSPSR` become `UNKNOWN`.

For more information on the operation of the DCPS<n> instructions, see [DCPS](#).



Encoding

DCPS1

Decode for this encoding

```
// No additional decoding required.
```

Operation

```

if !Halted() then Undefined(); end;

if EL2Enabled() && PSTATE.EL == EL0 then
    let tge : bit = HCR().TGE;
    if tge == '1' then Undefined(); end;
end;

if PSTATE.EL != EL0 || ELUsingAArch32(EL1) then
    if PSTATE.EL != EL2 then
        AArch32_WriteMode(M32_Svc);
        PSTATE.E = SCTLR().EE;
        Rmode(14, M32_Svc) = ARBITRARY : bits(32);    // LR_svc
        SPSR_svc() = ARBITRARY : bits(32);
    else
        PSTATE.E = HSCTLR().EE;
        ELR_hyp() = ARBITRARY : bits(32);
        HSR() = ARBITRARY : bits(32);
        SPSR_hyp() = ARBITRARY : bits(32);
    end;

    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
end;

```

end;

`UpdateEDSCRFields();` // Update EDSCR PE state flags

C5.1.43 DCPS2

Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception level.

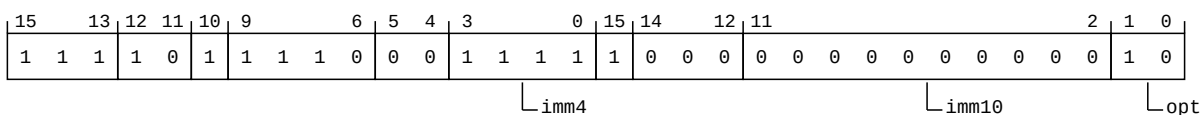
DCPS2 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL2 is not implemented.
- The PE is in Secure state and any of:
 - Secure EL2 is not implemented.
 - Secure EL2 is implemented and Secure EL2 is disabled.

When the PE executes DCPS2:

- If EL2 is using AArch32, the PE enters Hyp mode and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.
- If EL2 is using AArch64, the PE enters EL2 using AArch64, selects SP_EL2, and ELR_EL2, ESR_EL2, SPSR_EL2, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of the DCPS<n> instructions, see [DCPS](#).



Encoding

DCPS2

Decode for this encoding

```
if !HaveEL(EL2) then Undefined(); end;
```

Operation

```
if !Halted() || EDSCR().HDD == '1' then Undefined(); end;
```

```
if ELUsingAArch32(EL2) then
    AArch32_WriteMode(M32_Hyp);
    PSTATE.E = HSCTLR().EE;

    ELR_hyp() = ARBITRARY : bits(32);
    HSR() = ARBITRARY : bits(32);
    SPSR_hyp() = ARBITRARY : bits(32);

    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
```

end;

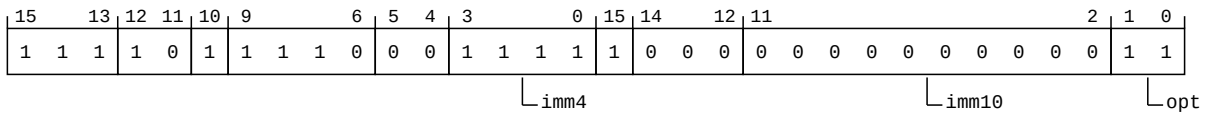
```
UpdateEDSCRFields(); // Update EDSCR PE state flags
```

C5.1.44 DCPS3

Debug Change PE State to EL3

This instruction is UNDEFINED.

For more information on the operation of the DCPS<n> instructions, see [DCPS](#).



Encoding

DCPS3

Decode for this encoding

```
if !HaveEL(EL3) then Undefined(); end;
```

Operation

```
if !Halted() then Undefined(); end;
```

C5.1.45 DFB

Data Full Barrier is a memory barrier that ensures the completion of memory accesses.

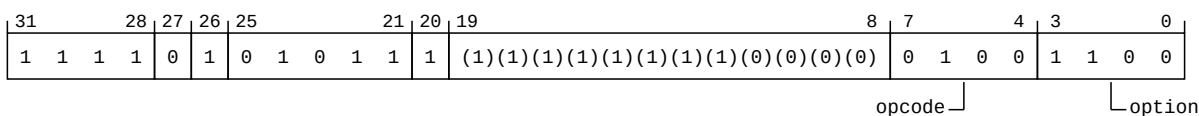
If executed at EL2, this instruction orders memory accesses irrespective of their Exception level or associated VMID. If executed at EL1 or EL0, this instruction behaves as DSB SY.

This is an alias of **DSB**. This means:

- The encodings in this description are named to match the encodings of [DSB](#).
- The description of [DSB](#) gives the operational pseudocode, any **CONSTRAINED UNPREDICTABLE** behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the A1 variant

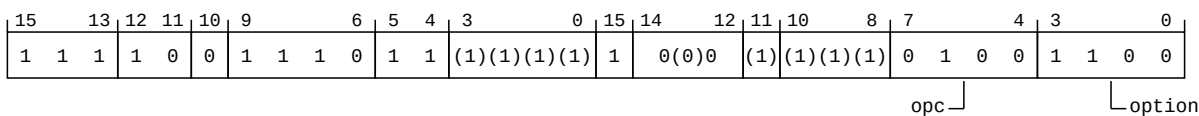
$$\text{DFB} \{ \langle c \rangle \} \{ \langle q \rangle \}$$

is equivalent to

DSB{<c>}{<q>} #12

and is always the preferred disassembly.

T1



Encoding for the T1 variant

$$\text{DFB} \{ \langle c \rangle \} \{ \langle q \rangle \}$$

is equivalent to

DSB{<c>}{<q>} #12

and is always the preferred disassembly.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). Must be AL or omitted.

For the “T1” variant: see [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

Operation

The description of [DSB](#) gives the operational pseudocode for this instruction.

C5.1.46 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see [Data Memory Barrier \(DMB\)](#).

The ordering requirements of Data Memory Barrier instruction is as follows:

- EL1 and EL0 memory accesses are ordered only with respect to memory accesses using the same VMID.
- EL2 memory accesses are ordered only with respect to other EL2 memory accesses.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	21	20	19	8	7	4	3	0
1	1	1	1	0	1	0	1	0	1	1	0	1
												option

└ opcode

Encoding

DMB{<c>}{<q>} {<option>}

Decode for this encoding

// No additional decoding required

T1

15	13	12	11	10	9	6	5	4	3	0	15	14	12	11	10	8	7	4	3	0								
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option

└opc

Encoding

DMB{<c>}{<q>} {<option>}

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). Must be `AL` or omitted.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<option>

Specifies the limitation on the Memory Effects ordered by the barrier operation, and is encoded in “option”.

Note

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

For more information on whether a Memory Effect is before or after a barrier instruction, see [Data Memory Barrier \(DMB\)](#).

option	<option>	Description
0001	OSHL D	This option has the same behavior as LD and is deprecated.
0010	OSHS T	This option has the same behavior as ST and is deprecated.
0011	OSH	This option has the same behavior as SY and is deprecated.
0101	NSHL D	This option has the same behavior as LD and is deprecated.
0110	NSHS T	This option has the same behavior as ST and is deprecated.
0111	NSH	This option has the same behavior as SY and is deprecated.
xx00	RESERVED	•
1001	ISHL D	This option has the same behavior as LD and is deprecated.
1010	ISHS T	This option has the same behavior as ST and is deprecated.
1011	ISH	This option has the same behavior as SY and is deprecated.
1101	LD	Reads are the required access type before the barrier instruction. Reads and writes are the required access types after the barrier instruction.
1110	ST	Writes are the required access type, both before and after the barrier instruction. <code>SYST</code> is a synonym for <code>ST</code> .

option	<option>	Description
1111	SY	Reads and writes are the required access types, both before and after the barrier instruction. Can be omitted.

All reserved options have the same behavior as SY, but software must not rely on this behavior.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    var types : MReqTypes;
    case option[1:0] of
        when '00' => types = MReqTypes_All;
        when '01' => types = MReqTypes_Reads;
        when '10' => types = MReqTypes_Writes;
        when '11' => types = MReqTypes_All;
    end;

    let vmid_sensitive : boolean = PSTATE.EL != EL2;
    DataMemoryBarrier(types, vmid_sensitive);
end;

```

C5.1.47 DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see [Data Synchronization Barrier \(DSB\)](#).

The ordering requirements of Data Synchronization Barrier instruction is as follows:

- EL1 and EL0 memory accesses are ordered only with respect to memory accesses using the same VMID.
- EL2 memory accesses are ordered only with respect to other EL2 memory accesses.

This instruction is used by the alias [DFB](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	21	20	19	8	7	4	3	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)(1)(1)(1)(1)(1)(1)(1)(0)(0)(0)(0)
0	1	0	0	option								

└ opcode

Encoding

DSB{<c>}{<q>}{<option>}

Decode for this encoding

// No additional decoding required

T1

15	13	12	11	10	9	6	5	4	3	0	15	14	12	11	10	8	7	4	3	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)(1)(1)(1)	1	0	(0)0	(1)(1)(1)(1)	0	1	0	0
option																				

└ opc

Encoding

DSB{<c>}{<q>}{<option>}

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). Must be `AL` or omitted.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<option>

Specifies the limitation on the Memory Effects and maintenance operations ordered by the barrier operation, and is encoded in “option”.

Note

The value `0b1100` is used to encode `DFB`.

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

For more information on whether an access is before or after a barrier instruction, see [Data Synchronization Barrier \(DSB\)](#).

option	<option>	Description
0001	OSHL	This option has the same behavior as LD and is deprecated.
0010	OSHST	This option has the same behavior as ST and is deprecated.
0011	OSH	This option has the same behavior as SY.
0x00	RESERVED	•
0101	NSHL	This option has the same behavior as LD and is deprecated.
0110	NSHST	This option has the same behavior as ST and is deprecated.
0111	NSH	Non-shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction.
1000	RESERVED	•
1001	ISHL	This option has the same behavior as LD and is deprecated.
1010	ISHST	This option has the same behavior as ST and is deprecated.
1011	ISH	Inner Shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction.
1101	LD	Reads are the required access type before the barrier instruction. Reads and writes are the required access types after the barrier instruction.
1110	ST	Writes are the required access type, both before and after the barrier instruction. <code>SYST</code> is a synonym for <code>ST</code> .
1111	SY	Outer Shareable is the required maintenance scope. Reads and writes are the required access types, both before and after the barrier instruction. Can be omitted.

All reserved options have the same behavior as SY, but software must not rely on this behavior.

Alias Conditions

Alias	Is preferred when
DFB	option == '1100'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let nXS : boolean = FALSE;
    var scope : MBMaintenanceScope;
    var types : MBReqTypes;
    case option[1:0] of
        when '00' =>
            types = MBReqTypes_All;
            scope = MBMaintenanceScope_OuterShareable;
        when '01' =>
            types = MBReqTypes_Reads;
            scope = MBMaintenanceScope_None;
        when '10' =>
            types = MBReqTypes_Writes;
            scope = MBMaintenanceScope_None;
        when '11' =>
            types = MBReqTypes_All;
            case option[3:2] of
                when '00' => scope = MBMaintenanceScope_OuterShareable;
                when '01' => scope = MBMaintenanceScope_Nonshareable;
                when '10' => scope = MBMaintenanceScope_InnerShareable;
                when '11' => scope = MBMaintenanceScope_OuterShareable;
            end;
    end;

    let vmid_sensitive : boolean = PSTATE.EL != EL2 && option[1:0] != '00';
    DataSynchronizationBarrier(scope, types, vmid_sensitive);
end;
```

C5.1.48 EOR, EORS (immediate)

Bitwise Exclusive-OR (immediate) performs a bitwise exclusive-OR of a register value and an immediate value, and writes the result to the destination register.

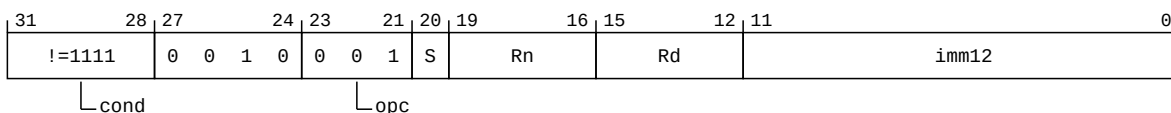
If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the EOR variant

Applies when (S == 0)

EOR{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the EORS variant

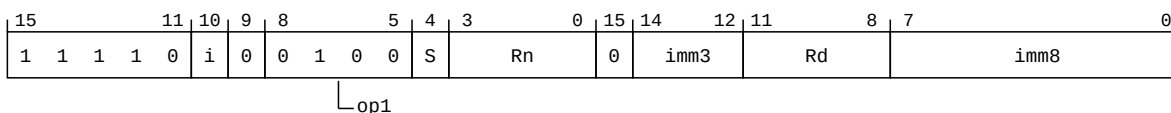
Applies when (S == 1)

EORS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding for the EOR variant

Applies when (S == 0)

EOR{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the EORS variant

Applies when (S == 1 && Rd != 1111)

EORS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("TEQ (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 EOR” and “A1 EORS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the EORS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the “T1 EOR” and “T1 EORS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 EOR” and “A1 EORS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 EOR” and “T1 EORS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 EOR” and “A1 EORS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1 EOR” and “T1 EORS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                             else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) XOR imm32;
    if d == 15 then                // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

C5.1.49 EOR, EORS (register)

Bitwise Exclusive-OR (register) performs a bitwise exclusive-OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

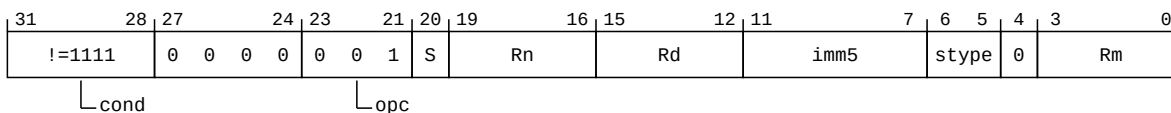
If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

The field descriptions for <RD> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The EOR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The EORS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the EOR, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

EOR {<c>} {<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the EOR, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

EOR{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the EORS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

EORS {<c>} {<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the EORS, shift or rotate by value variant

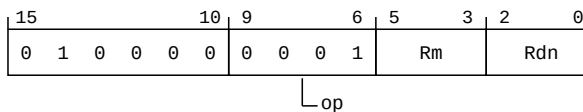
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

EORS{<c>}{<q>}{<Rd>,<Rn>,<Rm>{,<shift>#<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift t, shift n) : (SRTyp, integer) = DecodeImmShift(stype, imm5);
```

T1



Encoding

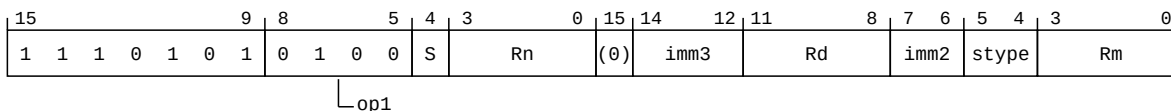
EOR<c>{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())

EORS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the EOR, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

EOR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the EOR, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

EOR{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

EOR<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the EORS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

EORS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the EORS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

EORS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

EORS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```

if Rd == '1111' && S == '1' then See("TEQ (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 EOR, rotate right with extend”, “A1 EOR, shift or rotate by value”, “A1 EORS, rotate right with extend”, and “A1 EORS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the EORS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T2 EOR, rotate right with extend”, “T2 EOR, shift or rotate by value”, “T2 EORS, rotate right with extend”, and “T2 EORS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 EOR, rotate right with extend”, “A1 EOR, shift or rotate by value”, “A1 EORS, rotate right with extend”, and “A1 EORS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 EOR, rotate right with extend”, “T2 EOR, shift or rotate by value”, “T2 EORS, rotate right with extend”, and “T2 EORS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 EOR, rotate right with extend”, “A1 EOR, shift or rotate by value”, “A1 EORS, rotate right with extend”, and “A1 EORS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 EOR, rotate right with extend”, “T2 EOR, shift or rotate by value”, “T2 EORS, rotate right with extend”, and “T2 EORS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 EOR, shift or rotate by value” and “A1 EORS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 EOR, shift or rotate by value” and “T2 EORS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

In T32 assembly:

- Outside an IT block, if EORS <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EORS <Rd>, <Rn> had been written
- Inside an IT block, if EOR<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though EOR<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .w qualifier.

Operation

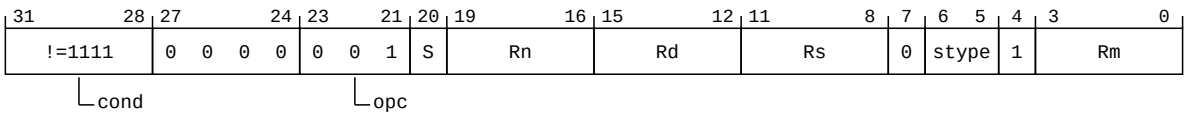
```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;

```

C5.1.50 EOR, EORS (register-shifted register)

Bitwise Exclusive-OR (register-shifted register) performs a bitwise exclusive-OR of a register value and a register-shifted register value. It writes the result to the destination register, and can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

EORS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

EOR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

C5.1.51 ERET

Exception Return.

The PE branches to the address held in the register holding the preferred return address, and restores **PSTATE** from SPSR <current mode>.

The register holding the preferred return address is:

- **ELR_hyp**, when executing in Hyp mode.
- **LR**, when executing in a mode other than Hyp mode, User mode, or System mode.

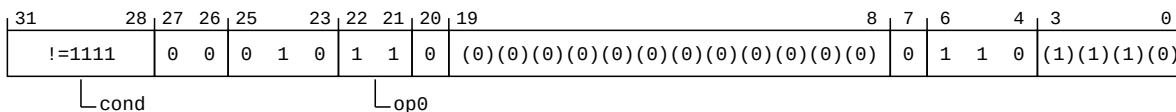
The PE checks SPSR <current mode> for an illegal return event. See [Illegal return events from AArch32 state](#).

Exception Return is CONSTRAINED UNPREDICTABLE in User mode and System mode.

In Debug state, the T1 encoding of `ERET` executes the DRPS operation.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



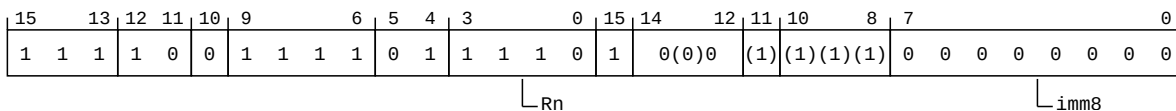
Encoding

ERET{ <c> } { <q> }

Decode for this encoding

```
// No additional decoding required
```

T1



Encoding

ERET { <c> } { <q> }

Decode for this encoding

```
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Halted() then
        if PSTATE.M IN {M32_User, M32_System} then
            UnpredictableProcedure();           // UNDEFINED or NOP
        else
            let new_pc_value : bits(32) = if PSTATE.EL == EL2 then ELR_hyp() else R(14);
            AArch32_ExceptionReturn(new_pc_value, SPSR_curr());
        end;
    else                                     // Perform DRPS operation in Debug state
        if PSTATE.M == M32_User then
            Undefined();
        elsif PSTATE.M == M32_System then
            UnpredictableProcedure();           // UNDEFINED or NOP
        else
            SynchronizeContext();
            DebugRestorePSR();
        end;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User, M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

C5.1.52 HLT

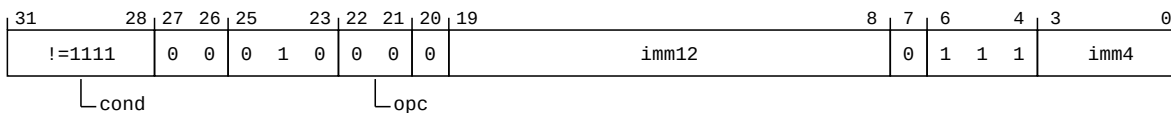
Halting Breakpoint

Halting breakpoint causes a software breakpoint to occur.

Halting breakpoint is always unconditional, even inside an IT block.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

HLT{<q>} {#}<imm>

Decode for this encoding

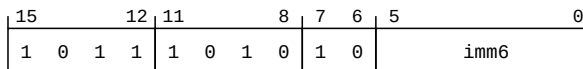
```
if EDSCR().HDE == '0' || !HaltingAllowed() then Undefined(); end;
if cond != '1110' then UnpredictableProcedure(); end; // HLT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1



Encoding

HLT{<q>} {#}<imm>

Decode for this encoding

```
if EDSCR().HDE == '0' || !HaltingAllowed() then Undefined(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#). An `HLT` instruction must be unconditional.

<imm>

For the “A1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm12:imm4” field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

For the “T1” variant: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the “imm6” field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

Operation

```
EncodingSpecificOperations();  
let is_async : boolean = FALSE;  
Halt(DebugHalt_HaltInstruction, is_async);
```

C5.1.53 HVC

Hypervisor Call causes a Hypervisor Call exception. For more information, see [Hypervisor Call \(HVC\) exception](#). Software executing at EL1 can use this instruction to call the hypervisor to request a service.

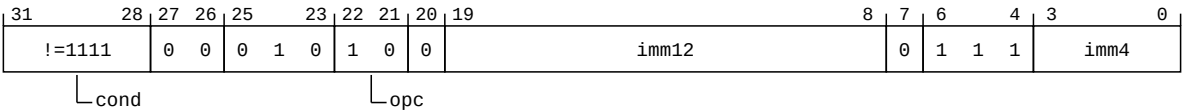
The HVC instruction is UNDEFINED:

- When [HCR.HCD](#) is set to 1.
- In User mode.

On executing an HVC instruction, the [HSR, Hyp Syndrome Register](#) reports the exception as a Hypervisor Call exception, using the EC value 0x12, and captures the value of the immediate argument, see [Use of the HSR](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

HVC{<q>} {#}<imm16>

Decode for this encoding

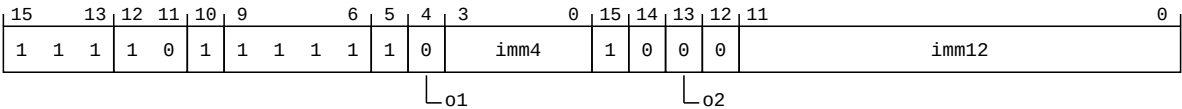
```
if cond != '1110' then UnpredictableProcedure(); end;
let imm16 : bits(16) = imm12::imm4;
```

CONSTRAINED UNPREDICTABLE behavior

If cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1



Encoding

HVC{<q>} {#}<imm16>

Decode for this encoding

```
let imm16 : bits(16) = imm4::imm12;
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#). An HVC instruction must be unconditional.

<imm16>

For the “A1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm12:imm4” field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

For the “T1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:imm12” field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

Operation

```
EncodingSpecificOperations();
if PSTATE.EL IN {EL0, EL3} || !EL2Enabled() then
    Undefined();
end;

var hvc_enable : bit;
hvc_enable = NOT(HCR().HCD);

if hvc_enable == '0' then
    Undefined();
else
    AArch32_CallHypervisor(imm16);
end;
```

CONSTRAINED UNPREDICTABLE behavior

If ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

C5.1.54 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see [Instruction Synchronization Barrier \(ISB\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	21	20	19	8	7	4	3	0
1	1	1	1	0	1	0	1	0	1	1	0	option

└ opcode

Encoding

ISB{<c>}{<q>}{<option>}

Decode for this encoding

// No additional decoding required

T1

15	13	12	11	10	9		6	5	4	3		0	15	14		12	11	10		8	7			4	3		0
1	1	1	1	0	0	1	1	1	0	1	1	(1)(1)(1)(1)	1	0(0)0		(1)	(1)(1)(1)(1)	0	1	1	0			option			

└opc

Encoding

ISB{<c>}{<q>}{<option>}

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). Must be `AL` or omitted.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<option>

Specifies an optional limitation on the barrier operation. Values are:

SY Full system barrier operation, encoded as `option = 0b1111`. Can be omitted.

All other encodings of <option> are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
end;
```

C5.1.55 IT

If-Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block are the same as, or the inverse of, the condition the `IT` instruction specifies for the first instruction in the block.

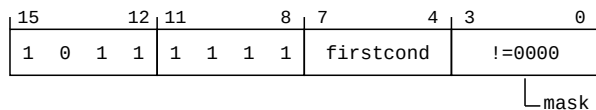
The `IT` instruction itself does not affect the condition flags, but the execution of the instructions in the IT block can change the condition flags.

16-bit instructions in the IT block, other than `CMP`, `CMN` and `TST`, do not set the condition flags. An `IT` instruction with the `AL` condition can change the behavior without conditional execution.

The architecture permits exception return to an instruction in the IT block only if the restoration of the `CPSR` restores `PSTATE.IT` to a state consistent with the conditions specified by the `IT` instruction. Any other exception return to an instruction in an IT block is UNPREDICTABLE. Any branch to a target instruction in an IT block is not permitted, and if such a branch is made it is UNPREDICTABLE what condition is used when executing that target instruction and any subsequent instruction in the IT block.

An implementation might include ITD controls that can disable the use of `IT` for some instructions, making them UNDEFINED.

For more information see [Conditional execution](#) and [Conditional instructions](#). The first of these sections includes more information about the ITD controls.



Encoding

`IT{<x>{<y>{<z>}}}{<q>} <cond>`

Decode for this encoding

```
if mask == '0000' then See("Related encodings"); end;
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then
    UnpredictableProcedure();
end;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The '1111' condition is treated as being the same as the '1110' condition, meaning always, and the `ITSTATE` state machine is progressed in the same way as for any other `cond_base` value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Miscellaneous 16-bit instructions](#).

Assembler Symbols

`<x>`

The condition for the second instruction in the IT block. If omitted, the “mask” field is set to `0b1000`. If present it is encoded in the “mask[3]” field:

T firstcond[0]
E NOT firstcond[0]

<y>

The condition for the third instruction in the IT block. If omitted and <x> is present, the “mask[2:0]” field is set to 0b100. If <y> is present it is encoded in the “mask[2]” field:

T firstcond[0]
E NOT firstcond[0]

<z>

The condition for the fourth instruction in the IT block. If omitted and <y> is present, the “mask[1:0]” field is set to 0b10. If <z> is present, the “mask[0]” field is set to 1, and it is encoded in the “mask[1]” field:

T firstcond[0]
E NOT firstcond[0]

<q>

See [Standard assembler syntax fields](#).

<cond>

The condition for the first instruction in the IT block, encoded in the “firstcond” field. See [Condition codes](#) for the range of conditions available, and the encodings.

The conditions specified in an **IT** instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check **IT** instruction syntax for validity but do not generate assembled instructions for them. See [Conditional instructions](#).

Operation

```
EncodingSpecificOperations();
AArch32_CheckITEnabled(mask);
PSTATE.IT[7:0] = firstcond::mask;
ShouldAdvanceIT = FALSE;
```

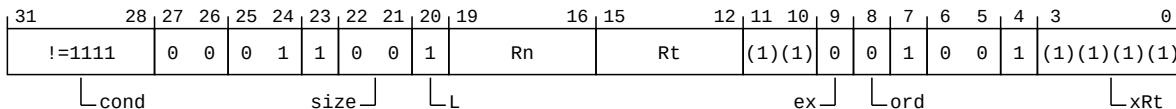
C5.1.56 LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#)

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



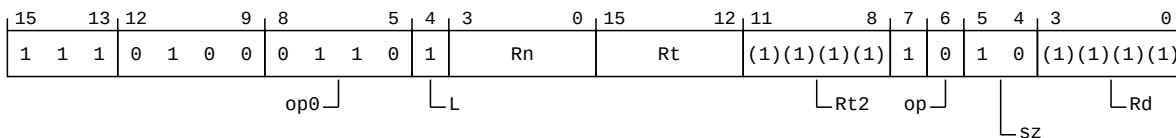
Encoding

LDA{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDA{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = MemO{32}(address);
end;
```

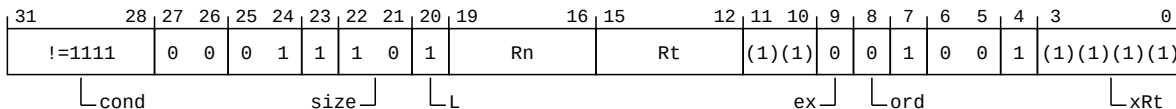
C5.1.57 LDAB

Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



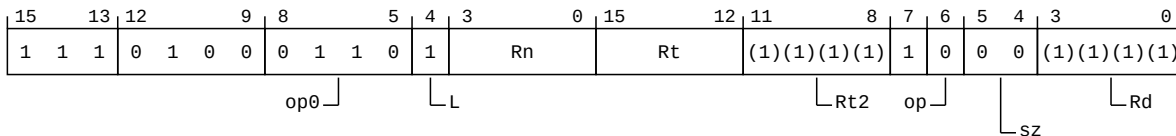
Encoding

LDAB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDAB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = ZeroExtend{32}(MemO{8}(address));
end;
```

C5.1.58 LDAEX

Load-Acquire Exclusive Word loads a word from memory, writes it to a register and:

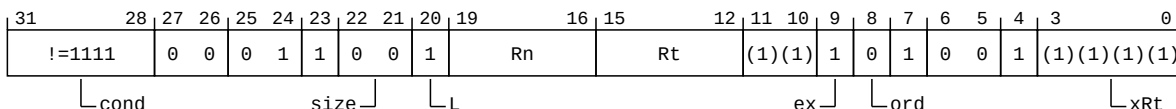
- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



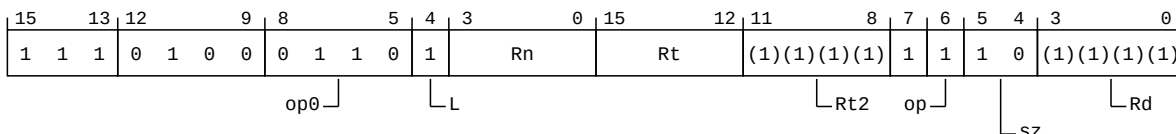
Encoding

LDAEX{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDAEX{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 4);
    R(t) = MemO{32}(address);
end;
```

C5.1.59 LDAEXB

Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

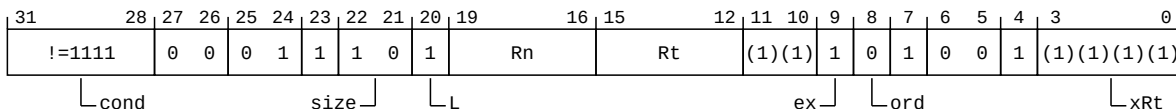
- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



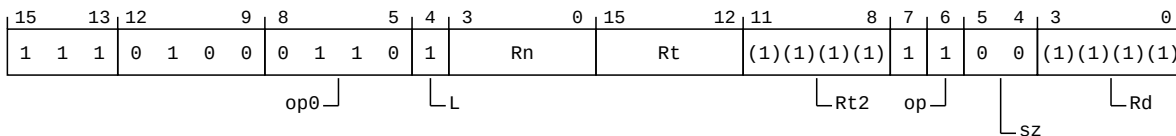
Encoding

LDAEXB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDAEXB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 1);
    R(t) = ZeroExtend{32}(MemO{8}(address));
end;
```

C5.1.60 LDAEXD

Load-Acquire Exclusive Doubleword loads a doubleword from memory, writes it to two registers and:

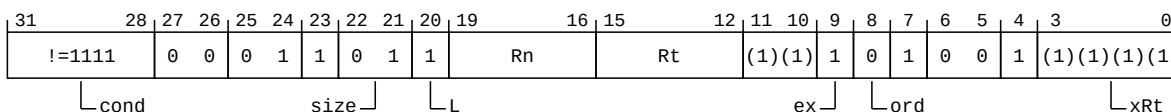
- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also acts as a barrier instruction with the ordering requirements described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
if Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

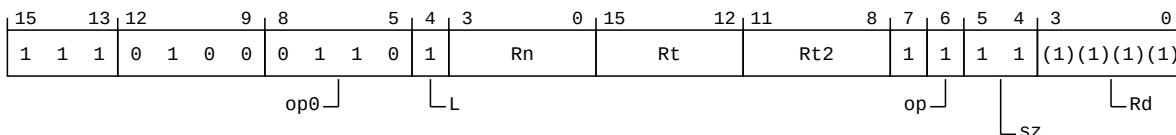
If Rt<0> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If Rt == '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15.

T1



Encoding

LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $t == t2$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. <Rt> must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. <Rt2> must be $R(t+1)$.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 8);
    let value : bits(64) = MemO{64}(address);
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R(t)  = if BigEndian(AccessType_GPR) then value[63:32] else value[31:0];
    R(t2) = if BigEndian(AccessType_GPR) then value[31:0]  else value[63:32];
end;
```

C5.1.61 LDAEXH

Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

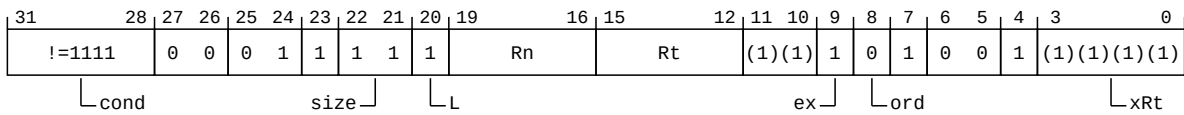
- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



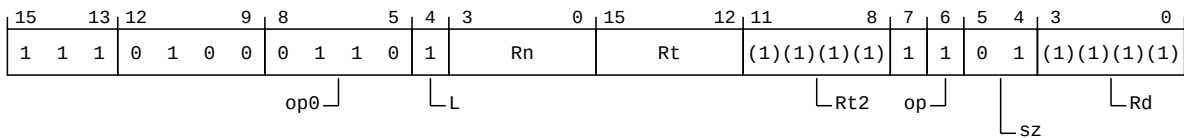
Encoding

LDAEXH{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDAEXH{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 2);
    R(t) = ZeroExtend{32}(MemO{16}(address));
end;
```

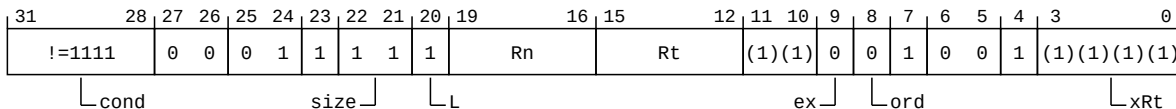
C5.1.62 LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



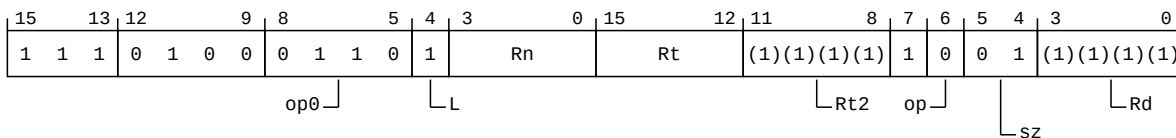
Encoding

LDAH{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDAH{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    R(t) = ZeroExtend{32}(MemO{16}(address));
end;
```

C5.1.63 LDC (immediate)

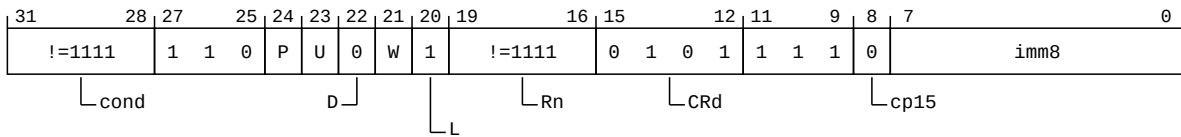
Load data to System register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to the [DBGDTRTXint](#) System register. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses, see [Memory accesses](#).

In an implementation that includes EL2, the permitted LDC access to [DBGDTRTXint](#) can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [HDCR.TDA](#).

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>{, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>, #<+/-><imm>]!

Encoding for the Unindexed variant

Applies when (P == 0 && U == 1 && W == 0)

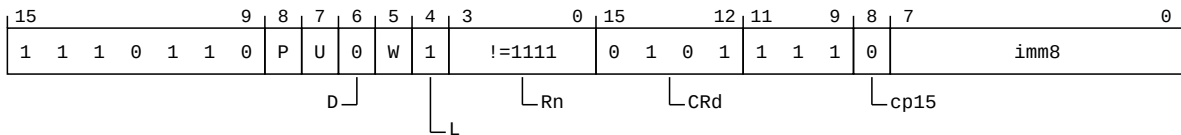
LDC{<c>}{<q>} p14, c5, [<Rn>], <option>

Decode for all variants of this encoding

```

if Rn == '1111' then See("LDC (literal)"); end;
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
```

T1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>{, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!

Encoding for the Unindexed variant

Applies when (P == 0 && U == 1 && W == 0)

LDC{<c>}{<q>} p14, c5, [<Rn>], <option>

Decode for all variants of this encoding

```

if Rn == 'l111' then See("LDC (literal)"); end;
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');

```

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

 $\langle R_n \rangle$

Is the general-purpose base register, encoded in the “Rn” field. If the PC is used, see [LDC \(literal\)](#).

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the “imm8” field, as <imm>/4.

<option>

Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the “imm8” field. The value of this field is ignored when executing this instruction.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);

    // System register write to DBGDTRTXint.
    AArch32_SysRegWriteM(cp, ThisInstr(), address);

    if wback then R(n) = offset_addr; end;
end;
```

C5.1.64 LDC (literal)

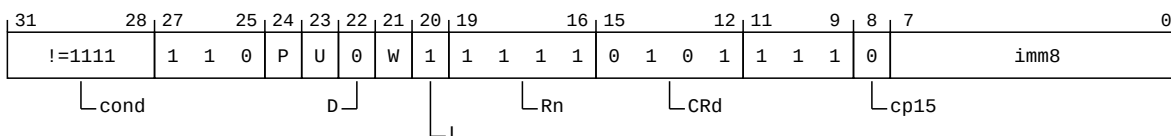
Load data to System register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to the **DBGDTRTXint** System register. For information about memory accesses, see [Memory accesses](#).

In an implementation that includes EL2, the permitted **LDC** access to **DBGDTRTXint** can be trapped to Hyp mode, meaning that an attempt to execute an **LDC** instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [HDCR.TDA](#).

For simplicity, the **LDC** pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



Encoding

Applies when $(!(P == '0' \ \&\& \ U == '0' \ \&\& \ W == '0'))$

LDC{<c>}{<q>} p14, c5, <label> // (Preferred syntax)

LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]

LDC{<c>}{<q>} p14, c5, [PC], <option>

Decode for this encoding

```

if P == '0' && U == '0' && W == '0' then Undefined(); end;
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;

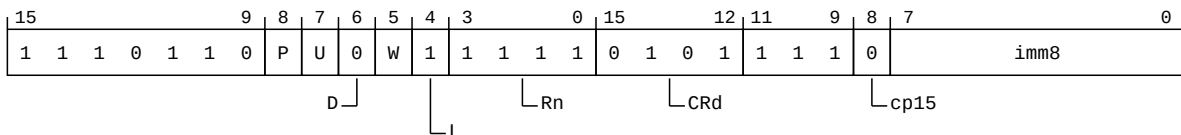
```

CONSTRAINED UNPREDICTABLE behavior

If $W == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as **NOF**.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1



Encoding

Applies when `(! (P == '0' && U == '0' && W == '0'))`

```
LDC{<c>}{<q>} p14, c5, <label> // (Preferred syntax)
```

LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]

Decode for this encoding

```
if P == '0' && U == '0' && W == '0' then Undefined(); end;
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet A32) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $W == '1' \parallel P == '0'$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes as `LDC (immediate)` with writeback to the PC. The instruction is handled as described in Using R15.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<label>

The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE` (encoded as `U == 1`).

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE` (encoded as `U == 0`).

 $+/ -$

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the “imm8” field, as <imm>/4.

<option>

Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the “imm8” field. The value of this field is ignored when executing this instruction.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                                else (AlignDownSize(PC32(),4) - imm32));
    let address : bits(32) = if index then offset_addr else AlignDownSize(PC32(),4);

    // System register write to DBGDTRTXint.
    AArch32_SysRegWriteM(cp, ThisInstr(), address);
end;
```

C5.1.65 LDM, LDMIA, LDMFD

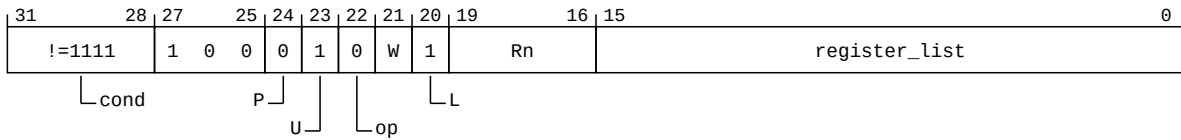
Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

This instruction is used by the alias **POP (multiple registers)**.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1** and **T2**).

A1



Encoding

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

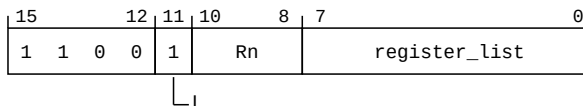
If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMPD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)

Decode for this encoding

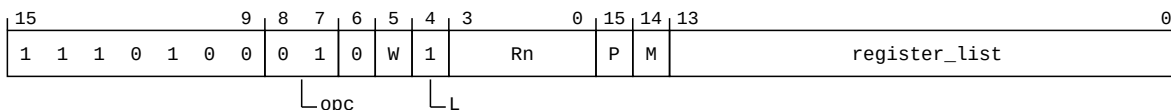
```
let n : integer = UInt(Rn);
let registers : bits(16) = '00000000'::register_list;
let wback : boolean = (registers[n] == '0');
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T2



Encoding

LDM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be
↪represented in T1)

LDMPD{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack, if <Rn>, '!' and
↪<registers> can be represented in T1)

LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMPD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

IA

Is an optional suffix for the Increment After form.

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

For the “A1” and “T2” variants: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

For the “T1” variant: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if <Rn> is included in <registers>, otherwise it must be present.

<registers>

For the “A1” variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For the “T1” variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the “register_list” field.

For the “T2” variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain one of the LR or the PC. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0. If the PC is in the list, the “P” field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Alias Conditions

Alias	Of variant	Is preferred when
POP (multiple registers)	A1	$W == '1' \ \&\& \ Rn == '1101' \ \&\& \ \text{BitCount}(\text{register_list}) > 1$
POP (multiple registers)	T2	$W == '1' \ \&\& \ Rn == '1101' \ \&\& \ \text{BitCount}(P : M : \text{register_list}) > 1$
POP (multiple registers)	T2	$\text{BitCount}(P : M : \text{register_list}) > 1$

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n);
    let length : integer = 4*BitCount(registers);
    let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address, within8byte);
            else
                data = MemS{32}(address, within8byte);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address, within8byte));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) + 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;

end;
```

C5.1.66 LDM (exception return)

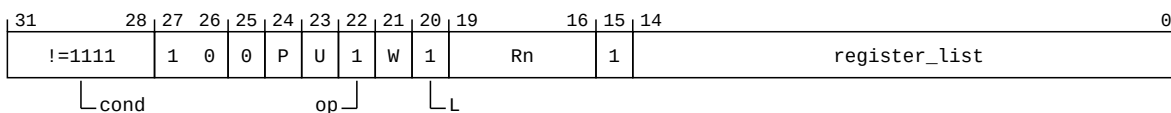
Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The [SPSR](#) of the current mode is copied to the [CPSR](#). An address adjusted by the size of the data loaded can optionally be written back to the base register.

The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

The PE checks the encoding that is copied to the [CPSR](#) for an illegal return event. See [Illegal return events from AArch32 state](#).

Load Multiple (exception return) is:

- UNDEFINED in Hyp mode.
- UNPREDICTABLE in debug state, and in User mode and System mode.



Encoding

LDM{<amode>}{<c>}{<q>} <Rn>{!}, <registers_with_pc>^

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(15) = register_list;
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && registers<n> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all the loads using the specified addressing mode and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during the execution of this instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<amode>

is one of:

- DA** Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
- FA** Full Ascending. For this instruction, a synonym for DA.
- DB** Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.

- EA** Empty Ascending. For this instruction, a synonym for **DB**.
- IA** Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
- FD** Full Descending. For this instruction, a synonym for **IA**.
- IB** Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
- ED** Empty Descending. For this instruction, a synonym for **IB**.

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers_with_pc>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC. See also [Encoding of lists of general-purpose registers and the PC](#).

Instructions with similar syntax but without the PC included in the registers list are described in [LDM \(User registers\)](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elseif PSTATE.M IN {M32_User, M32_System} then
        UnpredictableProcedure(); // UNDEFINED or NOP
    else
        let length : integer = 4*BitCount(registers) + 4;
        var address : bits(32) = if increment then R(n) else R(n)-length;
        let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
        if wordhigher then address = address+4; end;
        var data : bits(32);

        for i = 0 to 14 do
            if registers[i] == '1' then
                if i != n then
                    R(i) = MemS{32}(address, within8byte);
                else
                    data = MemS{32}(address, within8byte);
                end;
                address = address + 4;
            end;
        end;
        let new_pc_value : bits(32) = MemS{32}(address, within8byte);

        if wback && registers[n] == '1' then
            R(n) = ARBITRARY : bits(32);
        end;
```

```
        if wback && registers[n] == '0' then
            R(n) = if increment then R(n)+length else R(n)-length;
        end;
        if !wback && registers[n] == '1' then
            R(n) = data;
        end;

        AArch32_ExceptionReturn(new_pc_value, SPSR_curr());
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M` IN {`M32_User`, `M32_System`}, then one of the following behaviors must occur:

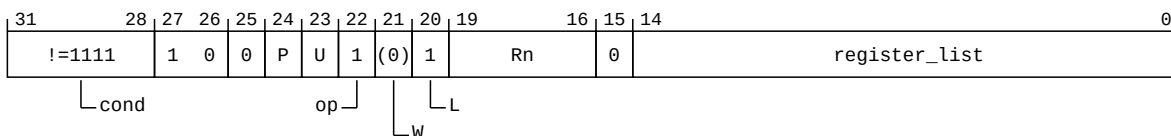
- The instruction is UNDEFINED.
- The instruction executes as NOP.

C5.1.67 LDM (User registers)

Load Multiple (User registers)

In an EL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Load Multiple (User registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.



Encoding

LDM{<amode>}{<c>}{<q>} <Rn>, <registers_without_pc>^

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(15) = register_list;
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<amode>

is one of:

- DA** Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
- FA** Full Ascending. For this instruction, a synonym for DA.
- DB** Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
- EA** Empty Ascending. For this instruction, a synonym for DB.
- IA** Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
- FD** Full Descending. For this instruction, a synonym for IA.
- IB** Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
- ED** Empty Descending. For this instruction, a synonym for IB.

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<registers_without_pc>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the LDM instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also [Encoding of lists of general-purpose registers and the PC](#).

Instructions with similar syntax but with the PC included in <registers_without_pc> are described in [LDM \(exception return\)](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then Undefined();
    elsif PSTATE.M IN {M32_User, M32_System} then UnpredictableProcedure();
    else
        let length : integer = 4*BitCount(registers);
        var address : bits(32) = if increment then R(n) else R(n)-length;
        let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
        if wordhigher then address = address+4; end;
        for i = 0 to 14 do
            if registers[i] == '1' then // Load User mode register
                Rmode(i, M32_User) = MemS{32}(address, within8byte); address = address + 4;
            end;
        end;
    end;
end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

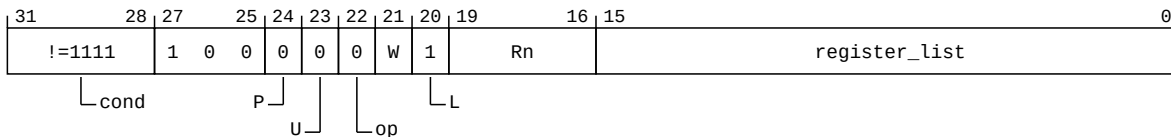
If PSTATE.M IN {M32_User, M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

C5.1.68 LDMDA, LDMFA

Load Multiple Decrement After (Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).



Encoding

LDMDA{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = (R(n) - 4*BitCount(registers)) + 4;
    let length : integer = 4*BitCount(registers);
    let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address, within8byte);
            else
                data = MemS{32}(address, within8byte);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address, within8byte));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) - 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;

end;

```

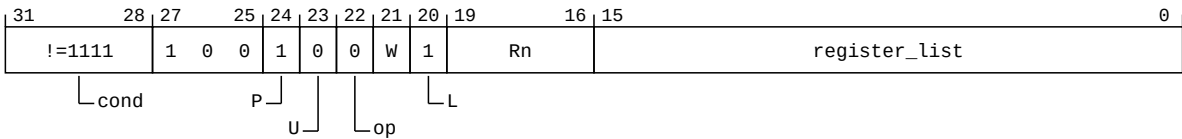
C5.1.69 LDMDB, LDMEA

Load Multiple Decrement Before (Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

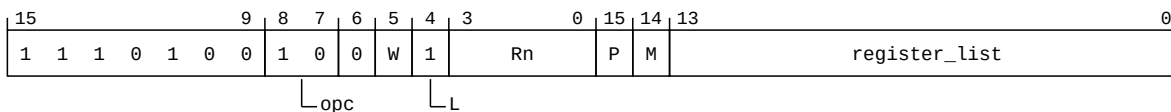
If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T1



Encoding

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```

let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include `R15`. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If registers<13> == '1', then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode, but R13 is **UNKNOWN**.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

For the “A1” variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

For the “T1” variant: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain one of the LR or the PC. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0. If the PC is in the list, the “P” field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) - 4*BitCount(registers);
    let length : integer = 4*BitCount(registers);
    let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address, within8byte);
            else
                data = MemS{32}(address, within8byte);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
```

```
        LoadWritePC(MemS{32}(address, within8byte));
    end;

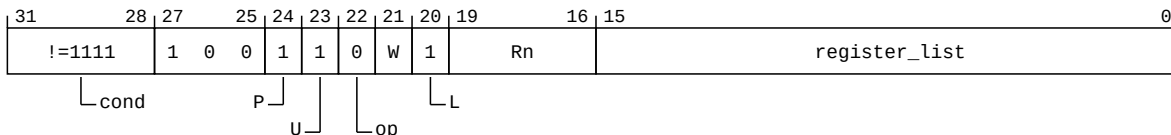
    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) - 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;

end;
```

C5.1.70 LDMIB, LDMED

Load Multiple Increment Before (Empty Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).



Encoding

```
LDMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
LDMED{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

Decode for this encoding

```

let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');

if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $\text{BitCount}(\text{registers}) < 1$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include `R15`. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) + 4;
    let length : integer = 4*BitCount(registers);
    let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
    var data : bits(32);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i != n then
                R(i) = MemS{32}(address, within8byte);
            else
                data = MemS{32}(address, within8byte);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        LoadWritePC(MemS{32}(address, within8byte));
    end;

    if wback && registers[n] == '1' then
        R(n) = ARBITRARY : bits(32);
    end;
    if wback && registers[n] == '0' then
        R(n) = R(n) + 4*BitCount(registers);
    end;
    if !wback && registers[n] == '1' then
        R(n) = data;
    end;

end;
```

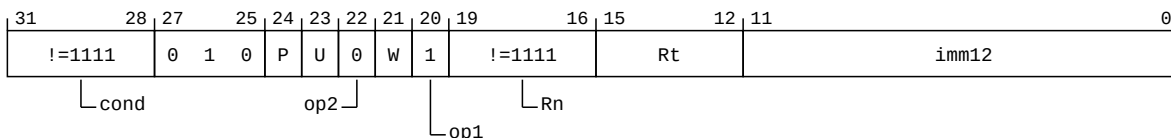
C5.1.71 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

This instruction is used by the alias [POP \(single register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), [T3](#), and [T4](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

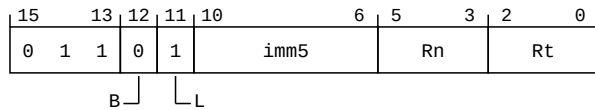
```
if Rn == '1111' then See("LDR (literal)"); end;
if P == '0' && W == '1' then See("LDRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm12)};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if wback && n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



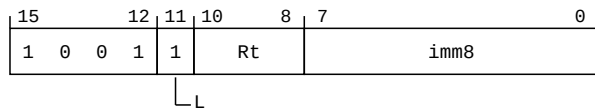
Encoding

LDR{<c>}{<q>} <Rt>, [<Rn> {, #}{+}<imm>]}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2



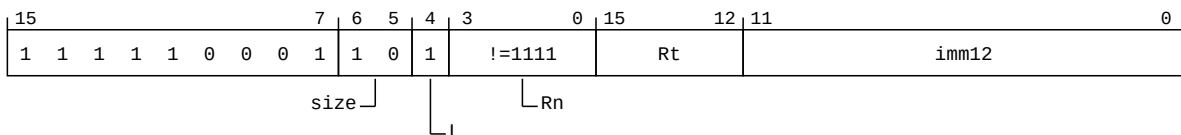
Encoding

LDR{<c>}{<q>} <Rt>, [SP{, #}{+}<imm>]}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = 13;
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T3



Encoding

LDR{<c>}{<q>} <Rt>, [<Rn> {, #}{+}<imm>]}

LDR{<c>}.W <Rt>, [<Rn> {, #}{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)

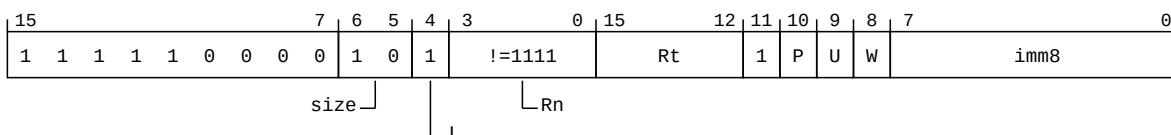
Decode for this encoding

```

if Rn == '1111' then See("LDR (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
if t == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;

```

T4



Encoding for the Offset variant

Applies when (P == 1 && U == 0 && W == 0)

```
LDR{<c>} {<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDR{<c>}{<q>} <Rt>, [<Rn>], # {+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDR{<c>}{<q>}<Rt>, [<Rn>, # {+/-}<imm>]!
```

Decode for all variants of this encoding

```

if Rn == '1111' then See("LDR (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then
  UnpredictableProcedure();
end;

```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T1” and “T2” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

For the “T3”, “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, “A1 Pre-indexed”, “T3”, “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. For PC use see [LDR \(literal\)](#).

For the “T1” variant: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the “imm5” field as `<imm>/4`.

For the “T2” variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the “imm8” field as <imm>/4.

For the “T3” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Alias Conditions

Alias	Of variant	Is preferred when
POP (single register)	A1 Post-indexed	P == '0' && U == '1' && W == '0' && Rn == '1101' && imm12 == '000000000100'
POP (single register)	T4 Post-indexed	Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100'

Operation

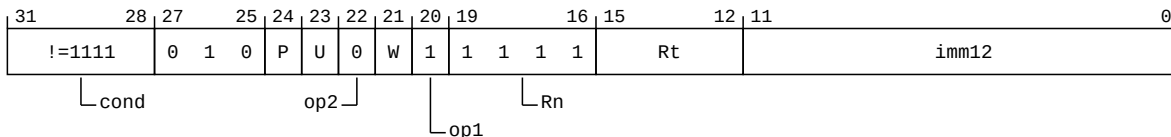
```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(32) = MemU{32}(address);
    if wback then R(n) = offset_addr; end;
    if t == 15 then
      if address[1:0] == '00' then
        LoadWritePC(data);
      else
        UnpredictableProcedure();
      end;
    else
      R(t) = data;
    end;
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(32) = MemU{32}(address);
    if wback then R(n) = offset_addr; end;
    if t == 15 then
      if address[1:0] == '00' then
        LoadWritePC(data);
      else
        UnpredictableProcedure();
      end;
    else
      R(t) = data;
    end;
  end;
end;
```

C5.1.72 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

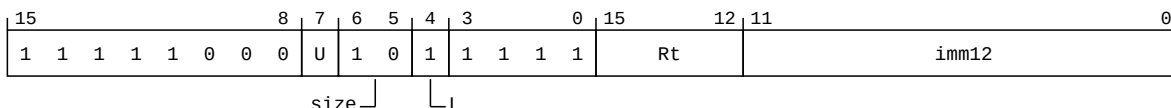
A1



Decode for this encoding

```
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let add : boolean = TRUE;
```

T2



Encoding

```
LDR{<c>}.W <Rt>, <label> // (Preferred syntax)
```

LDR{<c>}{<q>} <Rt>, <label> // (Preferred syntax, and <Rt>, <label> can be represented in T1)

```
LDR{<c>}{<q>} <Rt>, [PC, #(<+/-><imm>)] // (Alternative syntax)
```

Decode for this encoding

```

let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rt>

For the “A1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field.

For the “T2” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

<label>

For the “A1” and “T2” variants: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

For the “T1” variant: the label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.

`+/-`

Specifies the offset is added to or subtracted from the base register, defaulting to `+` if omitted and encoded in “U”:

U	+/-
0	-
1	+

`<imm>`

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “`imm12`” field.

For the “T2” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “`imm12`” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(), 4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(32) = MemU{32}(address);
    if t == 15 then
        if address[1:0] == '00' then
            LoadWritePC(data);
        else
            UnpredictableProcedure();
        end;
    else
        R(t) = data;
    end;
end;
```

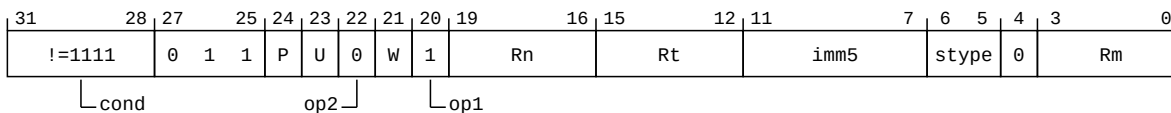
C5.1.73 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses, see [Memory accesses](#).

The T32 form of `LDR (register)` does not support register writeback.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDR{<c>}{<q>}<Rt>,[<Rn>,{+/-}<Rm>{,<shift>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!

Decode for all variants of this encoding

```

if P == '0' && W == '1' then See("LDRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;

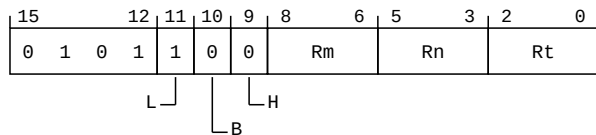
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

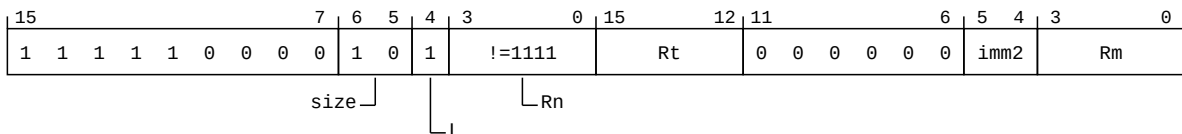
Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;

```

T2



Encoding

LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```

if Rn == '1111' then See("LDR (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if t == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This branch is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field.

For the “T2” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

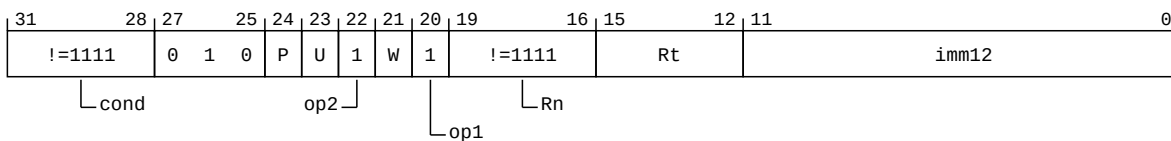
```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
        let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
        let address : bits(32) = if index then offset_addr else R(n);
        let data : bits(32) = MemU{32}(address);
        if wback then R(n) = offset_addr; end;
        if t == 15 then
            if address[1:0] == '00' then
                LoadWritePC(data);
            else
                UnpredictableProcedure();
            end;
        else
            R(t) = data;
        end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
        let offset_addr : bits(32) = (R(n) + offset);
        let address : bits(32) = offset_addr;
        let data : bits(32) = MemU{32}(address);
        if t == 15 then
            if address[1:0] == '00' then
                LoadWritePC(data);
            else
                UnpredictableProcedure();
            end;
        else
            R(t) = data;
        end;
    end;
end;
```

C5.1.74 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

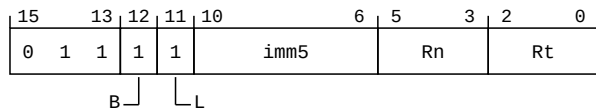
```
if Rn == '1111' then See("LDRB (literal)"); end;
if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

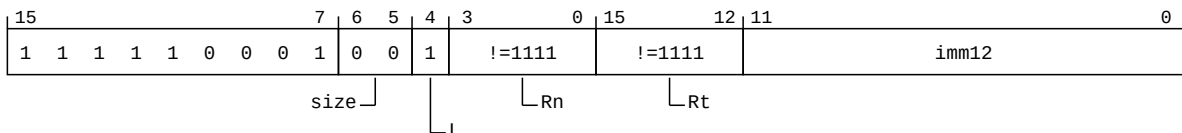
Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;

```

T2



Encoding

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

LDRB{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

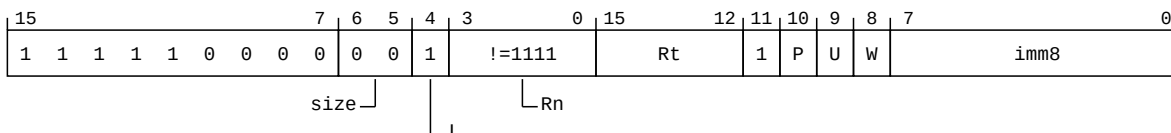
Decode for this encoding

```

if Rt == '1111' then See("PLD"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

```

T3



Encoding for the Offset variant

Applies when (Rt != 1111 && P == 1 && U == 0 && W == 0)

LDRB{<c>}{<q>}<Rt>,[<Rn>{,<#->imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDRB{<c>}{<q>} <Rt>, [<Rn>], # {+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!
```

Decode for all variants of this encoding

```

if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLD, PLDW (immediate)"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRBT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13

```

CONSTRAINED UNPREDICTABLE behavior

If $w_{back} \ \&\& \ n == t$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is `UNKNOWN`. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, “A1 Pre-indexed”, “T2”, “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. For PC use see [LDRB \(literal\)](#).

For the “T1” variant: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.

For the “T2” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = ZeroExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
```

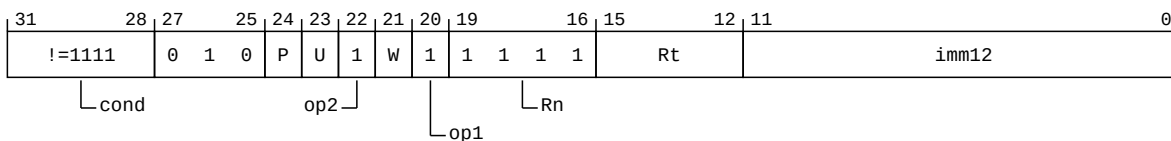
```
    R(t) = ZeroExtend{32}(MemU{8}(address));  
    if wback then R(n) = offset_addr; end;  
end;  
end;
```

C5.1.75 LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

Applies when $(!(P == '0' \ \&\& \ W == '1'))$

LDRB{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)

Decode for this encoding

```

if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;

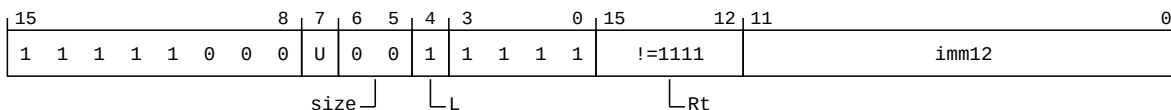
```

CONSTRAINED UNPREDICTABLE behavior

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRB (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1



Encoding

LDRB{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)

Decode for this encoding

```
if Rt == '1111' then See("PLD"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<label>

The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

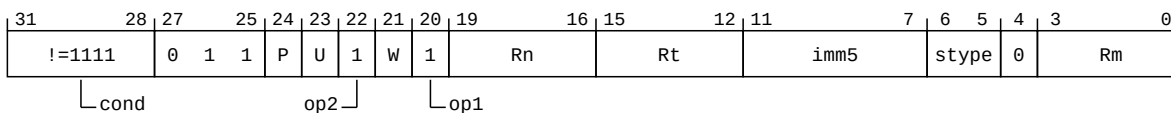
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(), 4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    R(t) = ZeroExtend{32}(MemU{8}(address));
end;
```

C5.1.76 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}]
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

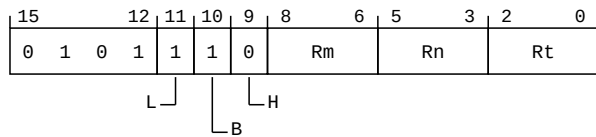
```
if P == '0' && W == '1' then See("LDRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



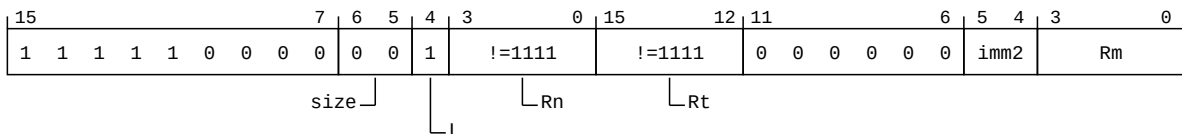
Encoding

LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTtype = SRTtype_LSL;
let shift_n : integer = 0;
```

T2



Encoding

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```

if Rt == '1111' then See("PLD"); end;
if Rn == '1111' then See("LDRB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = ZeroExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
end;
```

C5.1.77 LDRBT

Load Register Byte Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

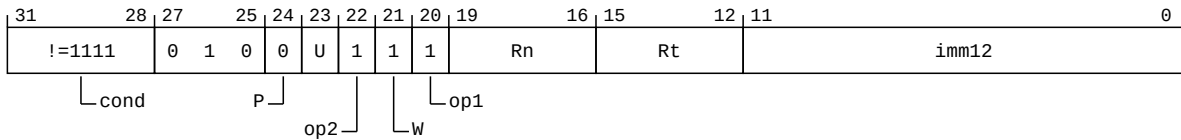
LDRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1



Encoding

LDRBT{<C>}{<Q>} <Rt>, [<Rn>] {, #(<+/->)<imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

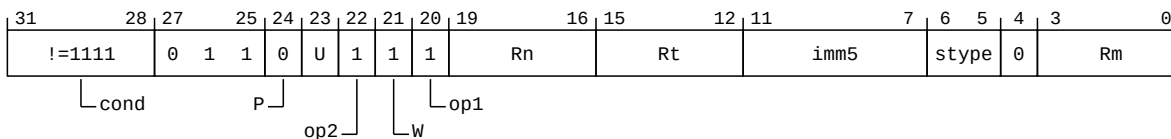
If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction uses immediate offset addressing with the base register as PC, without writeback.

If $n == t$ && $n != 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



Encoding

LDRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;

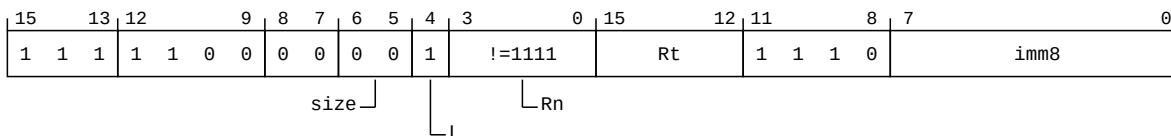
```

CONSTRAINED UNPREDICTABLE behavior

If $n == t$ && $n != 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRBT{<c>}{<q>} <Rt>, [<Rn> {, #+}<imm>}]

Decode for this encoding

```
if Rn == '1111' then See("LDRB (literal)"); end;  
let t : integer = UInt(Rt);  
let n : integer = UInt(Rn);  
let postindex : boolean = FALSE;  
let add : boolean = TRUE;  
let register_form : boolean = FALSE;  
let imm32 : bits(32) = ZeroExtend{}(imm8);  
let m : integer = ARBITRARY : integer;  
let shift_n : integer = ARBITRARY : integer;  
let shift_t : SRTYPE = ARBITRARY : SRTYPE;  
// Armv8-A removes UNPREDICTABLE for R13  
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “A2” and “T1” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).

+

Specifies the offset is added to the base register.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;           // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                             else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    R(t) = ZeroExtend{32}(MemU_unpriv{8}(address));
    if postindex then R(n) = offset_addr; end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

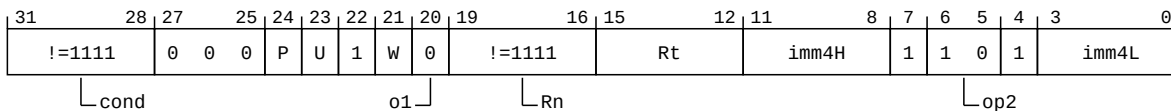
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRB (immediate).

C5.1.78 LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #<+/-><imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #<+/-><imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #<+/-><imm>]!
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRD (literal)"); end;
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
if t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

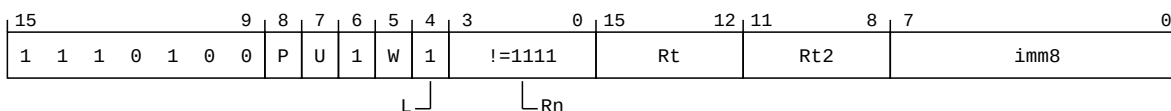
If P == '0' && W == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If $R_{t<0} == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $t<0> = '0'$.
- The instruction executes with the additional decode: $t2 = t$.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when $Rt == '1111'$.

T1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRD{<c>}{<q>}<Rt>, <Rt2>, [<Rn> {, # {+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

```
LDRD{<c>}{<q>}    <Rt>, <Rt2>, [<Rn>], #<+/-><imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRD{<c>}{<q>}    <Rt>, <Rt2>, [<Rn>, #<+/-><imm>]!
```

Decode for all variants of this encoding

```

if P == '0' && W == '0' then See("Related encodings"); end;
if Rn == '1111' then See("LDRD (literal)"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The load instruction executes but the destination register takes an **UNKNOWN** value.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Load/store dual](#), [load/store exclusive](#), [table branch](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the second general-purpose register to be transferred. This register must be `<R (t+1)>`.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. For PC use see [LDRD \(literal\)](#).

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the “imm8” field as <imm>/4.

Operation

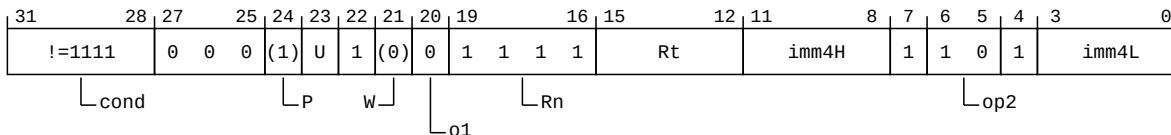
```
if ConditionPassed() then
    EncodingSpecificOperations();
let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
let address : bits(32) = if index then offset_addr else R(n);
if IsAlignedSize(address, 8) then
    let data : bits(64) = MemA{64}(address);
    if BigEndian(AccessType_GPR) then
        R(t)  = data[63:32];
        R(t2) = data[31:0];
    else
        R(t)  = data[31:0];
        R(t2) = data[63:32];
    end;
else
    let data1 : bits(32) = MemA{32}(address);
    let data2 : bits(32) = MemA{32}(address+4);
    R(t)  = data1;
    R(t2) = data2;
end;
if wback then R(n) = offset_addr; end;
end;
```

C5.1.79 LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)
```

```
LDRD{<c>}{<q>}    <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```

if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let add : boolean = (U == '1');
if t2 == 15 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

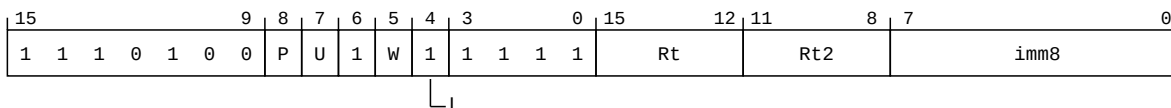
If $R_{t<0} == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $t<0> = '0'$.
- The instruction executes with the additional decode: $t2 = t$.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when $Rt == '1111'$.

If $P == '0' \parallel W == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as if $P = 1$ and $W = 0$.

T1



Encoding

Applies when $(!(P == '0' \ \&\& \ W == '0'))$

LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)

Decode for this encoding

```
if P == '0' && W == '0' then See("Related encodings"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let add : boolean = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
if W == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $t == t_2$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

If $W == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses post-indexed addressing when $P == '0'$ and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Load/Store dual](#), [Load/Store-Exclusive](#), [Load-Acquire/Store-Release](#), [table branch](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. This register must be $\langle R(t+1) \rangle$.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<label>

For the “A1” variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`. If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

For the “T1” variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	–
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

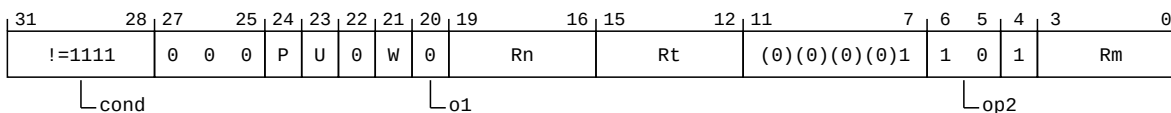
Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                                else (AlignDownSize(PC32(),4) - imm32));
    if IsAlignedSize(address, 8) then
        let data : bits(64) = MemA{64}(address);
        if BigEndian(AccessType_GPR) then
            R(t)  = data[63:32];
            R(t2) = data[31:0];
        else
            R(t)  = data[31:0];
            R(t2) = data[63:32];
        end;
    else
        let data1 : bits(32) = MemA{32}(address);
        let data2 : bits(32) = MemA{32}(address+4);
        R(t)  = data1;
        R(t2) = data2;
    end;
end;
```


C5.1.80 LDRD (register)

Load Register Dual (register) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if t2 == 15 || m == 15 || m == t || m == t2 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

If P == '0' && W == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If $m == t \mid \mid m == t2$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads register Rm with an UNKNOWN value.

If $Rt < 0 > == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $t < 0 > = '0'$.
- The instruction executes with the additional decode: $t2 = t$.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when $Rt == '1111'$.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

<Rt2>

Is the second general-purpose register to be transferred. This register must be $<R(t+1)>$.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + R(m)) else (R(n) - R(m));
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        let data : bits(64) = MemA{64}(address);
        if BigEndian(AccessType_GPR) then
            R(t)  = data[63:32];
            R(t2) = data[31:0];
        else
            R(t)  = data[31:0];
            R(t2) = data[63:32];
        end;
    else
        let data1 : bits(32) = MemA{32}(address);
        let data2 : bits(32) = MemA{32}(address+4);
        R(t)  = data1;
        R(t2) = data2;
    end;

    if wback then R(n) = offset_addr; end;
end;
```

C5.1.81 LDREX

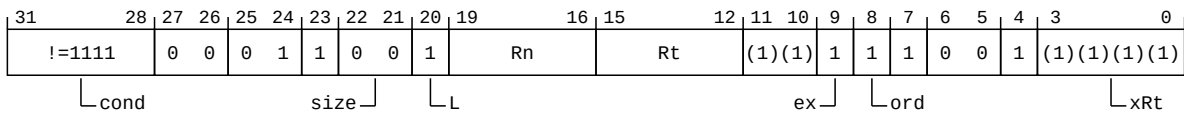
Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



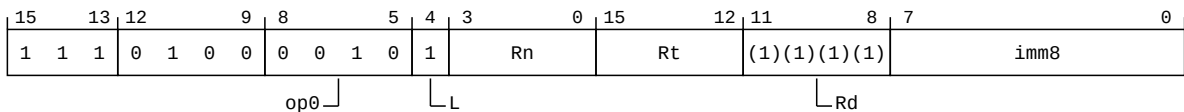
Encoding

LDREX{<c>}{<q>} <Rt>, [<Rn> {, {#}<imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = Zeros{}; // Zero offset
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<imm>

For the “A1” variant: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted.

For the “T1” variant: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n) + imm32;
    AArch32_SetExclusiveMonitors(address, 4);
    R(t) = MemA{32}(address);
end;
```

C5.1.82 LDREXB

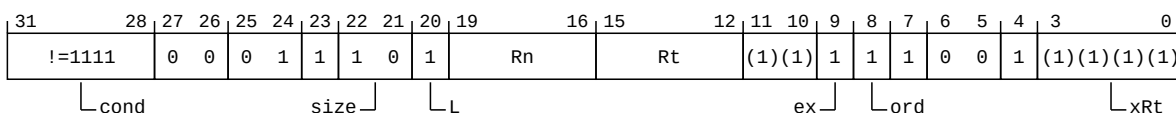
Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



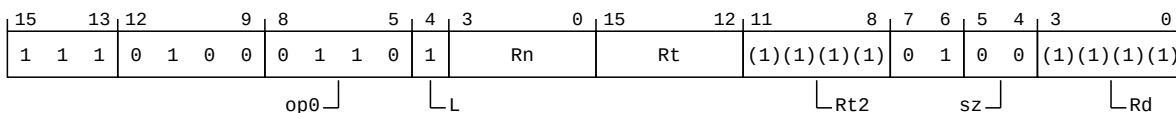
Encoding

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 1);
    R(t) = ZeroExtend{32}(MemA{8}(address));
end;
```

C5.1.83 LDREXD

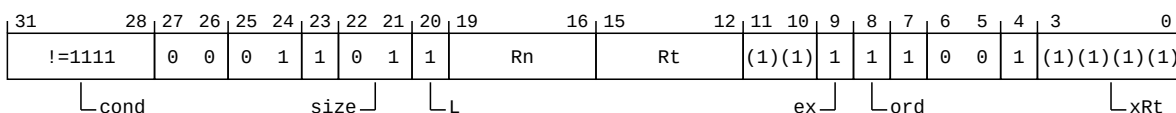
Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
if Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

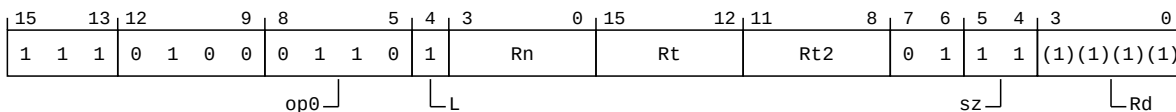
If Rt<0> == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If Rt == '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15.

T1



Encoding

LDRDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If $t == t2$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. <Rt> must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. <Rt2> must be $\langle R(t+1) \rangle$.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address,8);
    let value : bits(64) = MemA{64}(address);

    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R(t)  = if BigEndian(AccessType_GPR) then value[63:32] else value[31:0];
    R(t2) = if BigEndian(AccessType_GPR) then value[31:0]  else value[63:32];
end;
```

C5.1.84 LDREXH

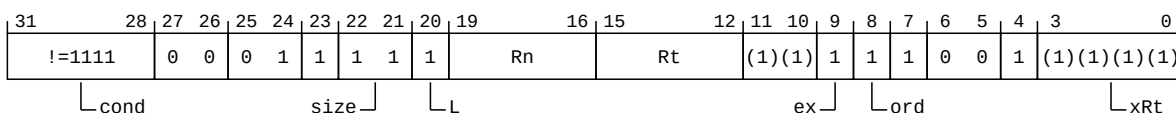
Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



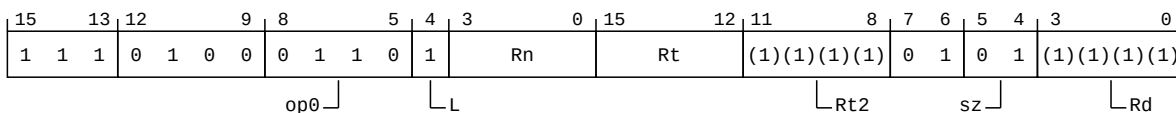
Encoding

LDREXH{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

LDREXH{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

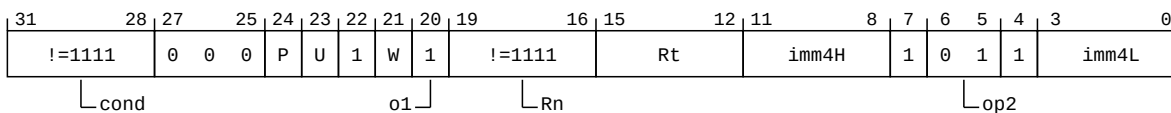
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    AArch32_SetExclusiveMonitors(address, 2);
    R(t) = ZeroExtend{32}(MemA{16}(address));
end;
```

C5.1.85 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

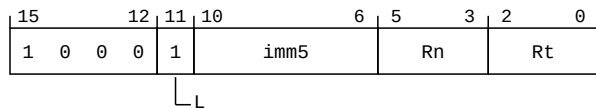
```
if Rn == '1111' then See("LDRH (literal)"); end;
if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



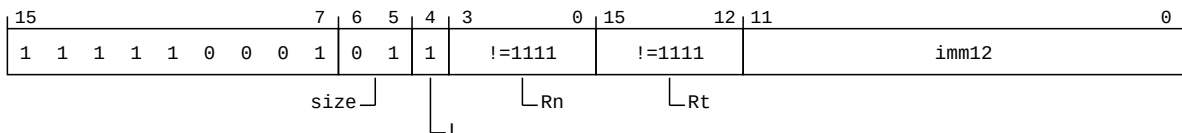
Encoding

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5::'0');
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2



Encoding

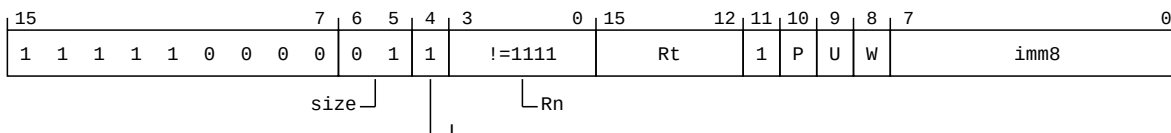
```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

```
LDRH{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```
if Rt == '1111' then See("PLD (immediate)"); end;
if Rn == '1111' then See("LDRH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3



Encoding for the Offset variant

Applies when (Rt != 1111 && P == 1 && U == 0 && W == 0)

LDRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

Decode for all variants of this encoding

```

if Rn == '1111' then See("LDRH (literal)"); end;
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLDW (immediate)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRHT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13

```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, “A1 Pre-indexed”, “T2”, “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. For PC use see [LDRH \(literal\)](#).

For the “T1” variant: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the “imm5” field as <imm>/2.

For the “T2” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = ZeroExtend{32}(data);
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
```



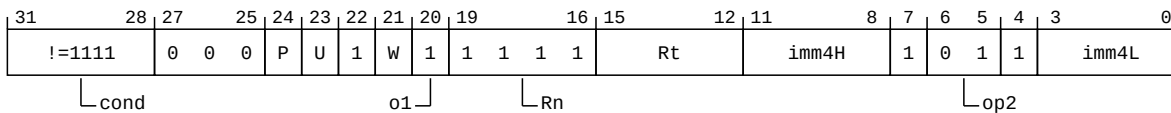
```
        let address : bits(32) = if index then offset_addr else R(n);
        let data : bits(16) = MemU{16}(address);
        if wback then R(n) = offset_addr; end;
        R(t) = ZeroExtend{32}(data);
    end;
end;
```

C5.1.86 LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

Applies when $(!(P == '0' \ \&\& \ W == '1'))$

LDRH{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)

Decode for this encoding

```

if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;

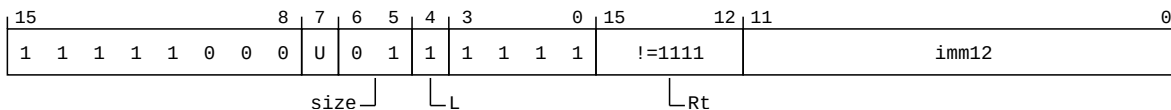
```

CONSTRAINED UNPREDICTABLE behavior

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRH (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1



Encoding

LDRH{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)

Decode for this encoding

```
if Rt == '1111' then See("PLD (literal)"); end;  
let t : integer = UInt(Rt);  
let imm32 : bits(32) = ZeroExtend{}(imm12);  
let add : boolean = (U == '1');  
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<label>

For the “A1” variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`. If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

For the “T1” variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

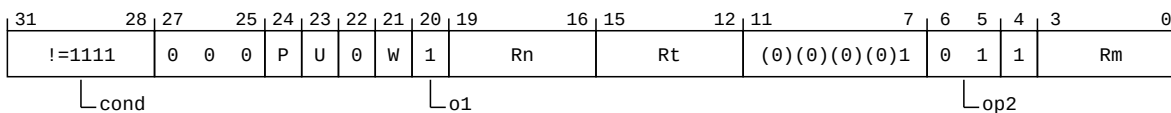
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(16) = MemU{16}(address);
    R(t) = ZeroExtend{32}(data);
end;
```

C5.1.87 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

$$\text{LDRH}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle Rt \rangle, \quad [\langle Rn \rangle, \quad \{+/-\}\langle Rm \rangle]$$

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

$$\text{LDRH}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle \text{Rt} \rangle, \quad [\langle \text{Rn} \rangle], \quad \{+/-\} \langle \text{Rm} \rangle$$

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

$$\text{LDRH}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle Rt \rangle, \quad [\langle Rn \rangle, \{+/-\}\langle Rm \rangle]!$$

Decode for all variants of this encoding

```

if P == '0' && W == '1' then See("LDRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRTtype = SRTtype_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;

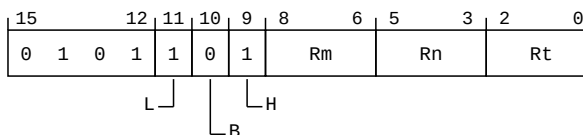
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

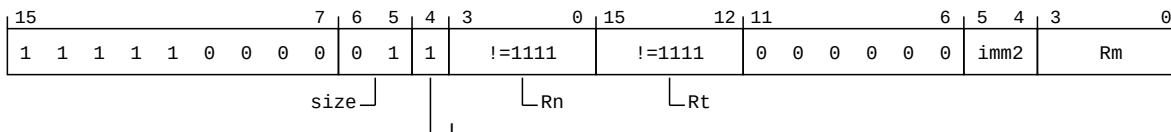
Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;

```

T2



Encoding

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```

if Rn == '1111' then See("LDRH (literal)"); end;
if Rt == '1111' then See("PLDW (register)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = ZeroExtend{32}(data);
end;
```

C5.1.88 LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

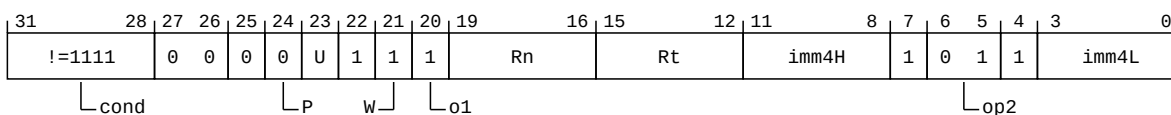
LDRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1



Encoding

LDRHT{<c>}{<q>} <Rt>, [<Rn>] {, #(<+/->)<imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

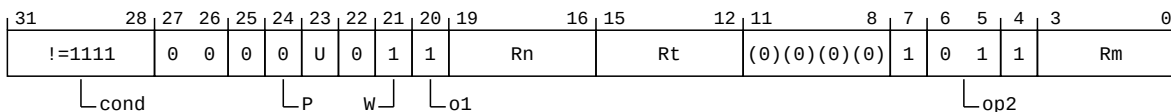
If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if $\text{bit}[24] == '1'$ and $\text{bit}[21] == '0'$. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If $n == t \ \&\& \ n \neq 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



Encoding

LDRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Decode for this encoding

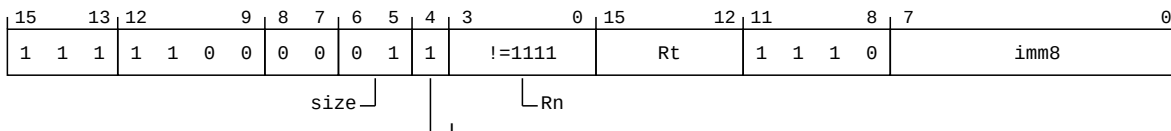
```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $n == t$ && $n != 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRHT{<c>}{<q>} <Rt>, [<Rn> {, #+}<imm>]

Decode for this encoding

```
if Rn == '1111' then See("LDRH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{ }(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;           // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    let data : bits(16) = MemU_unpriv{16}(address);
    if postindex then R(n) = offset_addr; end;
    R(t) = ZeroExtend{32}(data);
end;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

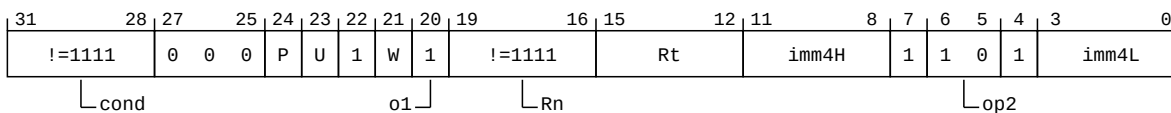
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRH (immediate).

C5.1.89 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRSB{<c>}{<q>} <Rt>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRSB{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```

if Rn == '1111' then See("LDRSB (literal)"); end;
if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;

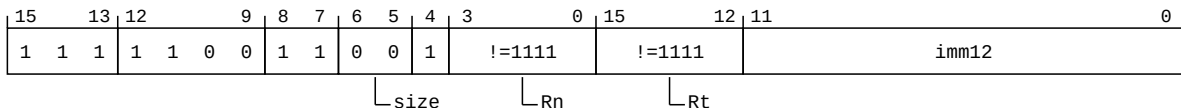
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

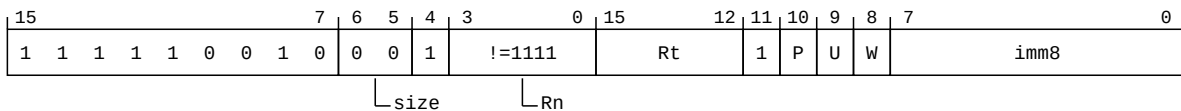
Decode for this encoding

```

if Rt == '1111' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

```

T2



Encoding for the Offset variant

Applies when (Rt != 1111 && P == 1 && U == 0 && W == 0)

LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDRSB{<c>}{<q>} <Rt>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRSB{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
if P == '1' && U == '1' && W == '0' then See("LDRSBT"); end;
if P == '0' && W == '0' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. For PC use see [LDRSB \(literal\)](#).

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T2 Offset”, “T2 Post-indexed”, and “T2 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

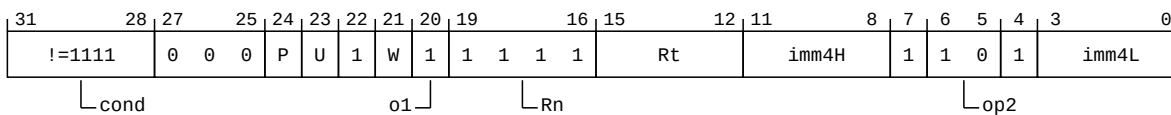
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = SignExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
end;
```

C5.1.90 LDRSB (literal)

Load Register Signed Byte (lrbt) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

Applies when `( ! (P == '0' && W == '1') )`

```
LDRSB{<c>}{<q>}    <Rt>, <label> // (Normal form)
```

```
LDRSB{<c>}{<q>}    <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```

if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;

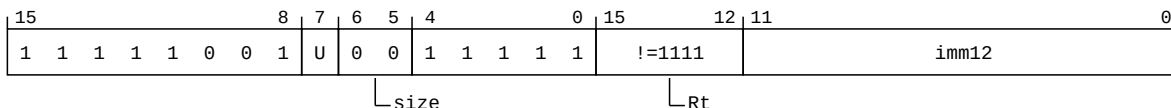
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRSB (immediate). The instruction uses post-indexed addressing when P = '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1



Encoding

```
LDRSB{<c>}{<q>} <Rt>, <label> // (Preferred syntax)
```

```
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```


Decode for this encoding

```
if Rt == '1111' then See("PLI"); end;  
let t : integer = UInt(Rt);  
let imm32 : bits(32) = ZeroExtend{}(imm12);  
let add : boolean = (U == '1');  
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<label>

For the “A1” variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`. If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

For the “T1” variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

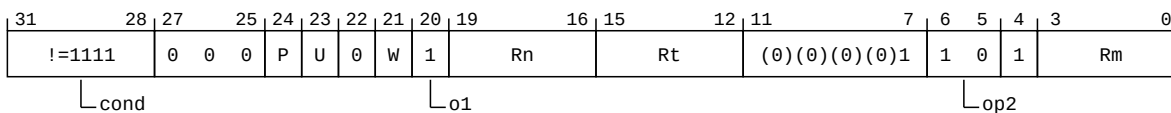
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    R(t) = SignExtend{32}(MemU{8}(address));
end;
```

C5.1.91 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

$$\text{LDRSB}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle Rt \rangle, [\langle Rn \rangle, \{+/-\}\langle Rm \rangle]$$

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRSB{<c>}{<q>}<Rt>,[<Rn>],{+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRSB{<c>}{<q>}<Rt>,[<Rn>,{+/-}<Rm>]!

Decode for all variants of this encoding

```

if P == '0' && W == '1' then See("LDRSBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRTtype = SRTtype_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;

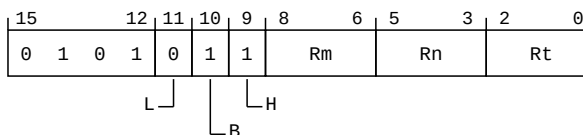
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



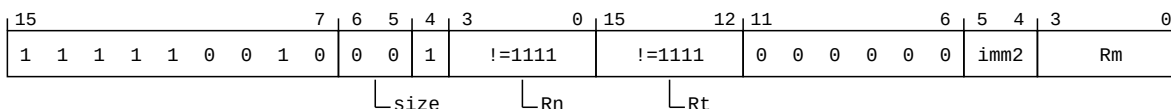
Encoding

LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```
if Rt == '1111' then See("PLI"); end;
if Rn == '1111' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    R(t) = SignExtend{32}(MemU{8}(address));
    if wback then R(n) = offset_addr; end;
end;
```

C5.1.92 LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

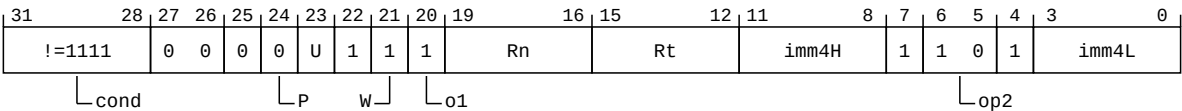
LDRSBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1



Encoding

LDRSBT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

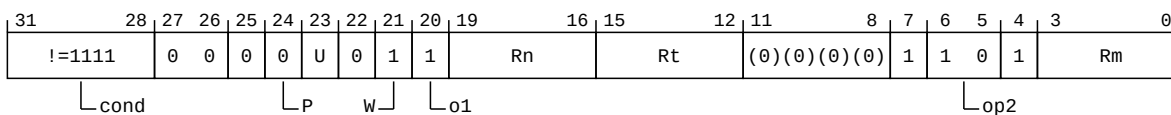
If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if `bit[24] == '1'` and `bit[21] == '0'`. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



Encoding

$$\text{LDRSBT}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle Rt \rangle, \quad [\langle Rn \rangle], \quad \{+/-\}\langle Rm \rangle$$

Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;

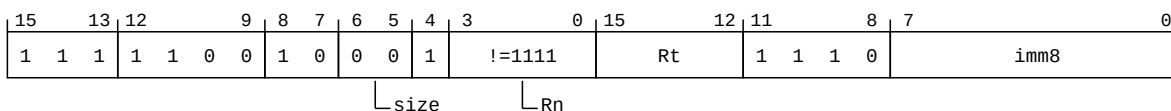
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t` && `n != 15`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRSBT{<c>}{<q>}<Rt>,[<Rn>{, # {+<imm>}]

Decode for this encoding

```

if Rn == 'l1l1' then See("LDRSB (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
if PSTATE.EL == EL2 then UnpredictableProcedure(); end;           // Hyp mode
let offset : bits(32) = if register_form then R(m) else imm32;
```



```
let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);  
let address : bits(32) = if postindex then R(n) else offset_addr;  
R(t) = SignExtend{32}(MemU_unpriv{8}(address));  
if postindex then R(n) = offset_addr; end;  
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

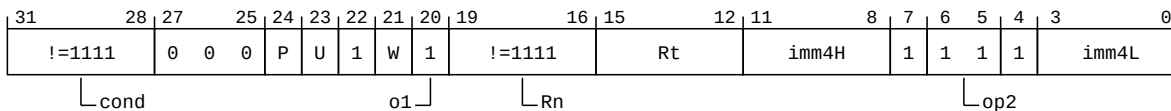
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRSB (immediate).

C5.1.93 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, # {+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], # {+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, # {+/-}<imm>]!
```

Decode for all variants of this encoding

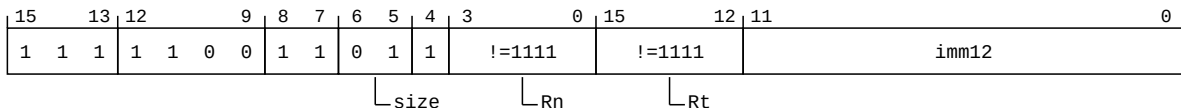
```
if Rn == '1111' then See("LDRSH (literal)"); end;
if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

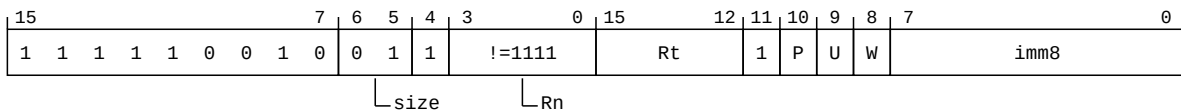
Decode for this encoding

```

if Rn == '1111' then See("LDRSH (literal)"); end;
if Rt == '1111' then See("Related instructions"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13

```

T2



Encoding for the Offset variant

Applies when (Rt != 1111 && P == 1 && U == 0 && W == 0)

LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

LDRSH{<c>}{<q>} <Rt>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

LDRSH{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;  
if Rt == '1111' && P == '1' && U == '0' && W == '0' then See("Related instructions"); end;  
if P == '1' && U == '1' && W == '0' then See("LDRSHT"); end;  
if P == '0' && W == '0' then Undefined(); end;  
let t : integer = UInt(Rt);  
let n : integer = UInt(Rn);  
let imm32 : bits(32) = ZeroExtend{}(imm8);  
let index : boolean = (P == '1');  
let add : boolean = (U == '1');  
let wback : boolean = (W == '1');  
if (t == 15 && W == '1') || (wback && n == t) then UnpredictableProcedure(); end;  
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related instructions: [Load/store single](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. For PC use see [LDRSH \(literal\)](#).

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T2 Offset”, “T2 Post-indexed”, and “T2 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

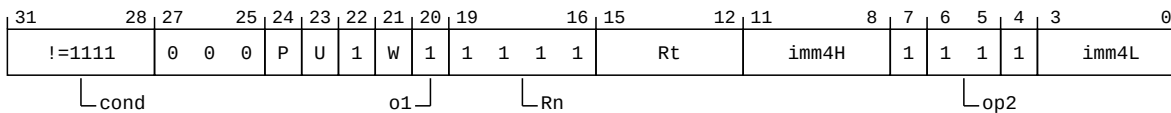
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = SignExtend{32}(data);
end;
```

C5.1.94 LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

Applies when $(!(P == '0' \ \&\& \ W == '1'))$

LDRSH{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)

Decode for this encoding

```

if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 || wback then UnpredictableProcedure(); end;

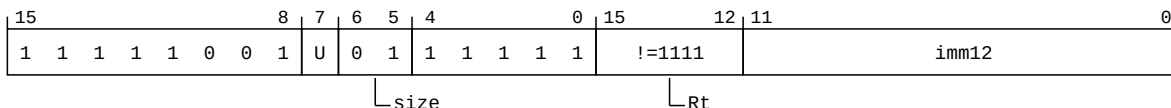
```

CONSTRAINED UNPREDICTABLE behavior

If wback, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in LDRSH (immediate). The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in Using R15.

T1



Encoding

LDRSH{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)

Decode for this encoding

```
if Rt == '1111' then See("Related instructions"); end;  
let t : integer = UInt(Rt);  
let imm32 : bits(32) = ZeroExtend{}(imm12);  
let add : boolean = (U == '1');  
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related instructions: [Load, signed \(literal\)](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<label>

For the “A1” variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Any value in the range -255 to 255 is permitted. If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`. If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

For the “T1” variant: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`, encoded as `U == 1`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`, encoded as `U == 0`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

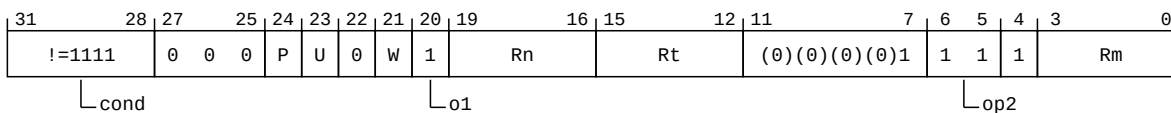
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = AlignDownSize(PC32(),4);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    let data : bits(16) = MemU{16}(address);
    R(t) = SignExtend{32}(data);
end;
```

C5.1.95 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

LDRSH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

Decode for all variants of this encoding

```

if P == '0' && W == '1' then See("LDRSHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRTtype = SRTtype_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;

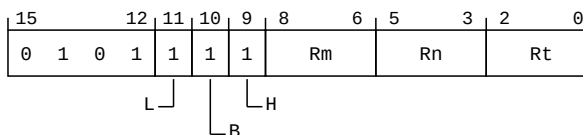
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is **UNKNOWN**. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

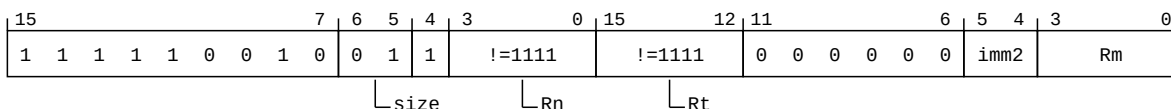
Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;

```

T2



Encoding

LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```

if Rn == '1111' then See("LDRSH (literal)"); end;
if Rt == '1111' then See("Related instructions"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related instructions: Load/store, signed (register offset).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    let data : bits(16) = MemU{16}(address);
    if wback then R(n) = offset_addr; end;
    R(t) = SignExtend{32}(data);
end;
```

C5.1.96 LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

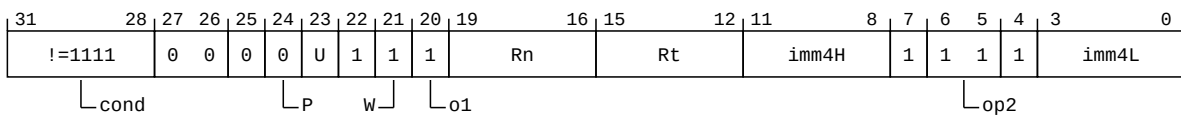
LDRSHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1



Encoding

LDRSHT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

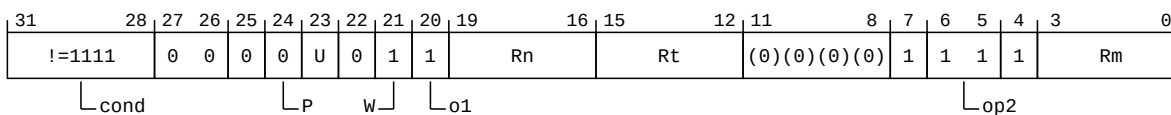
If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in [Using R15](#).
- The instruction is treated as if $\text{bit}[24] == '1'$ and $\text{bit}[21] == '0'$. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If $n == t \ \&\& \ n \neq 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



Encoding

LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Decode for this encoding

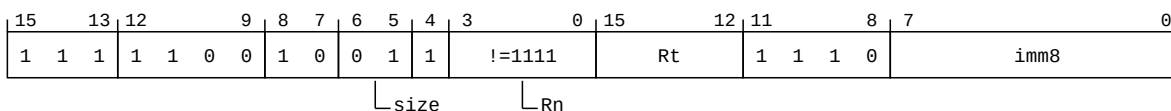
```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $n == t$ && $n != 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #+}<imm>]

Decode for this encoding

```
if Rn == '1111' then See("LDRSH (literal)"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
if PSTATE.EL == EL2 then UnpredictableProcedure(); end;           // Hyp mode
let offset : bits(32) = if register_form then R(m) else imm32;
```

```
let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);  
let address : bits(32) = if postindex then R(n) else offset_addr;  
let data : bits(16) = MemU_unpriv{16}(address);  
if postindex then R(n) = offset_addr; end;  
R(t) = SignExtend{32}(data);  
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDRSH (immediate).

C5.1.97 LDRT

Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

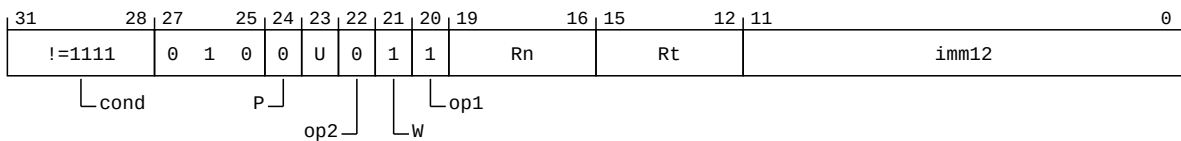
LDRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1



Encoding

LDRT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

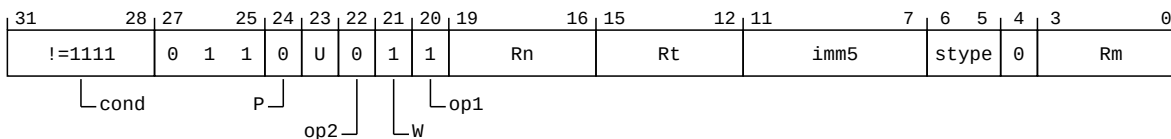
If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if $\text{bit}[24] == '1'$ and $\text{bit}[21] == '0'$. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If $n == t \ \&\& \ n \neq 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



Encoding

LDRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Decode for this encoding

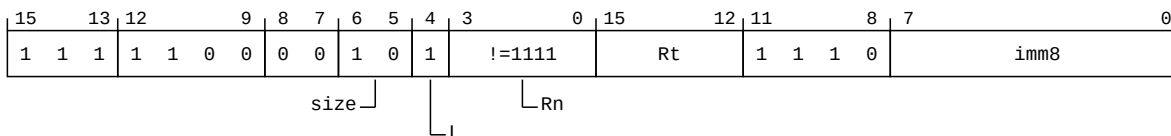
```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $n == t$ && $n != 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



Encoding

LDRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

Decode for this encoding

```
if Rn == '1111' then See("LDR (literal)"); end;  
let t : integer = UInt(Rt);  
let n : integer = UInt(Rn);  
let postindex : boolean = FALSE;  
let add : boolean = TRUE;  
let register_form : boolean = FALSE;  
let imm32 : bits(32) = ZeroExtend{}(imm8);  
let m : integer = ARBITRARY : integer;  
let shift_n : integer = ARBITRARY : integer;  
let shift_t : SRTYPE = ARBITRARY : SRTYPE;  
// Armv8-A removes UNPREDICTABLE for R13  
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “A2” and “T1” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).

+

Specifies the offset is added to the base register.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;           // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                             else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    let data : bits(32) = MemU_unpriv{32}(address);
    if postindex then R(n) = offset_addr; end;
    R(t) = data;
end;

```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDR (immediate).

C5.1.98 LSL (immediate)

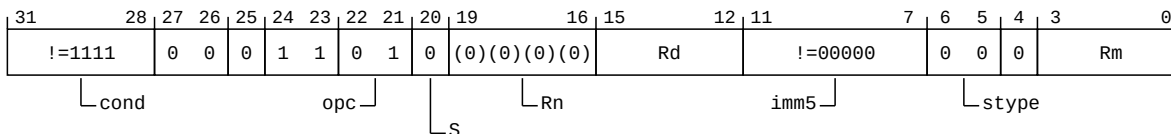
Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1



Encoding for the MOV, shift or rotate by value variant

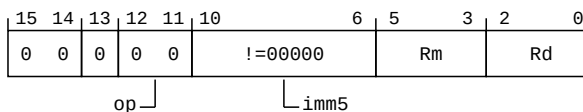
LSL{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>

and is always the preferred disassembly.

T2



Encoding for the T2 variant

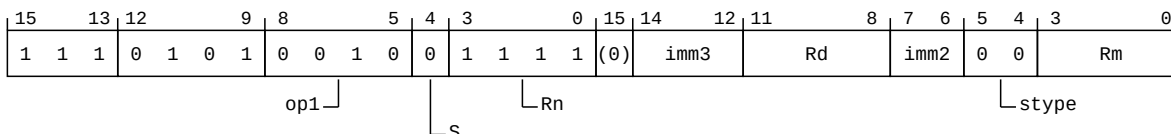
LSL<c>{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV<c>{<q>} <Rd>, <Rm>, LSL #<imm>

and is the preferred disassembly when `InITBlock()`.

T3



Encoding for the MOV, shift or rotate by value variant

Applies when $(!(\text{imm3} == 000 \ \&\& \ \text{imm2} == 00))$

```
LSL{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

```
LSL<c>.W {<Rd>}, <Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOV, shift or rotate by value” variant: is the shift amount, in the range 0 to 31, encoded in the “imm5” field as <imm> modulo 32.

For the “T2” variant: is the shift amount, in the range 1 to 31, encoded in the “imm5” field as <amount> modulo 32.

For the “T3 MOV, shift or rotate by value” variant: is the shift amount, in the range 0 to 31, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.99 LSL (register)

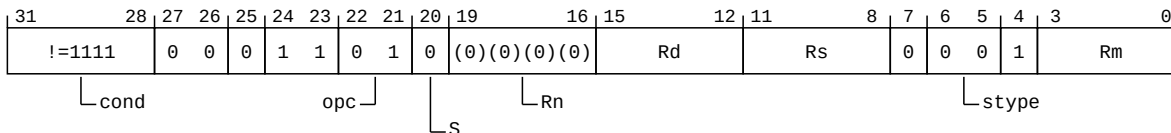
Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Not flag setting variant

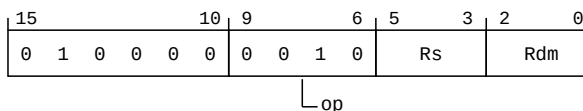
LSL{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>

and is always the preferred disassembly.

T1



Encoding for the Logical shift left variant

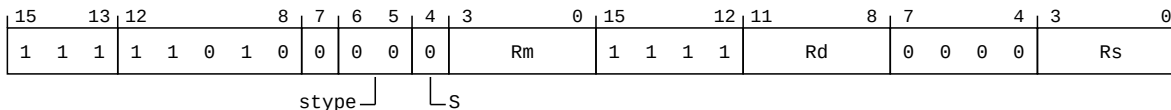
LSL<c>{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>

and is the preferred disassembly when `InITBlock()`.

T2



Encoding for the Not flag setting variant

```
LSL{<c>}{<q>}{<Rd>, }<Rm>, <Rs>
```

```
LSL<c>.W {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOV{<c>}{<q>}{<Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

C5.1.100 LSLS (immediate)

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

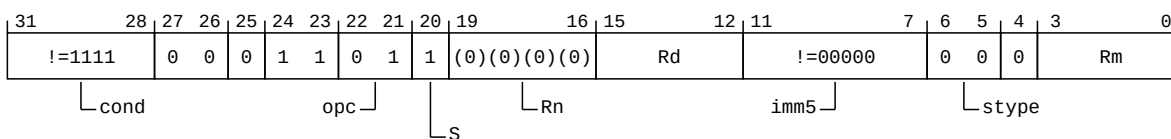
- The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1



Encoding for the MOVs, shift or rotate by value variant

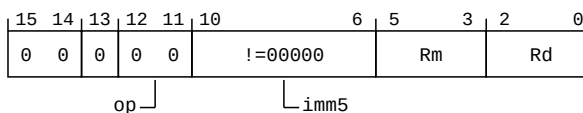
LSLS{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>

and is always the preferred disassembly.

T2



Encoding for the T2 variant

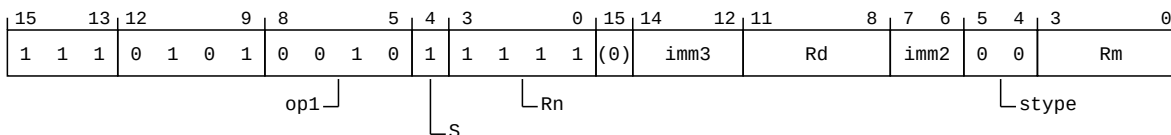
LSLS{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS<c>{<q>} <Rd>, <Rm>, LSL #<imm>

and is the preferred disassembly when !InITBlock().

T3



Encoding for the MOVS, shift or rotate by value variant

Applies when $(!(imm3 == 000 \ \&\& \ imm2 == 00))$

LSLS{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

LSLS.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOVS, shift or rotate by value” variant: is the shift amount, in the range 0 to 31, encoded in the “imm5” field as <imm> modulo 32.

For the “T2” variant: is the shift amount, in the range 1 to 31, encoded in the “imm5” field as <amount> modulo 32.

For the “T3 MOVS, shift or rotate by value” variant: is the shift amount, in the range 0 to 31, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of [MOV](#), [MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.101 LSLS (register)

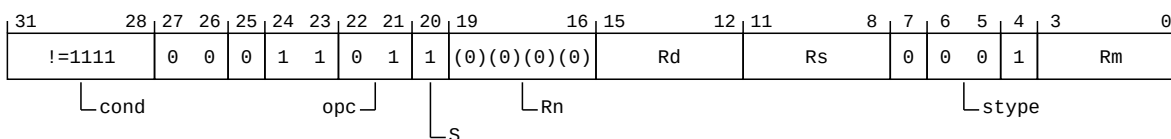
Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Flag setting variant

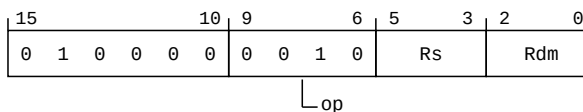
LSLS{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>

and is always the preferred disassembly.

T1



Encoding for the Logical shift left variant

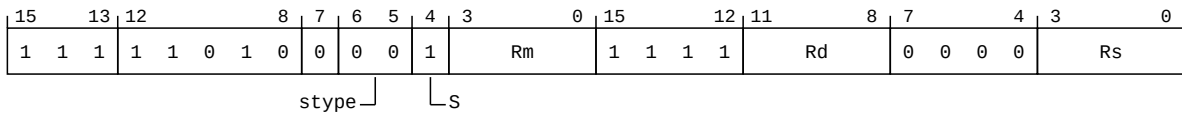
LSLS{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>

and is the preferred disassembly when !InITBlock().

T2



Encoding for the Flag setting variant

$$\text{LSLS}\{\langle c \rangle\}\{\langle q \rangle\} \quad \{\langle Rd \rangle, \} \langle Rm \rangle, \langle Rs \rangle$$

```
LSLS.W {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>

and is always the preferred disassembly.

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

 $\langle R_s \rangle$

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

C5.1.102 LSR (immediate)

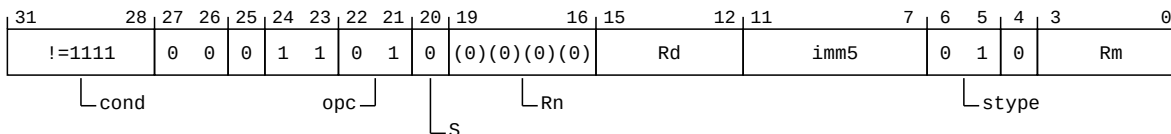
Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1



Encoding for the MOV, shift or rotate by value variant

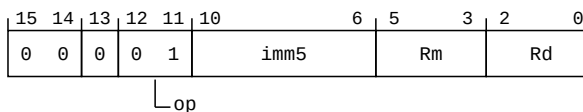
LSR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>

and is always the preferred disassembly.

T2



Encoding for the T2 variant

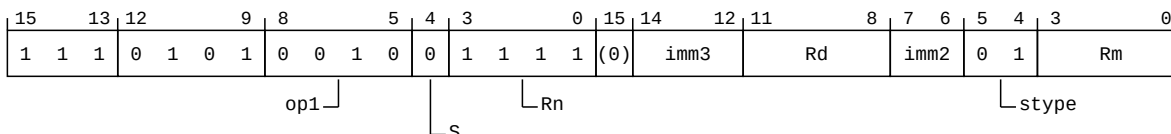
LSR<c>{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV<c>{<q>} <Rd>, <Rm>, LSR #<imm>

and is the preferred disassembly when `InITBlock()`.

T3



Encoding for the MOV, shift or rotate by value variant

```
LSR{<c>}{<q>}{<Rd>,<Rm>,<imm>}
```

```
LSR<c>.W {<Rd>,<Rm>,<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOV{<c>}{<q>}{<Rd>,<Rm>,<imm>}
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOV, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOV, shift or rotate by value” and “T2” variants: is the shift amount, in the range 1 to 32, encoded in the “imm5” field as <imm> modulo 32.

For the “T3 MOV, shift or rotate by value” variant: is the shift amount, in the range 1 to 32, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.103 LSR (register)

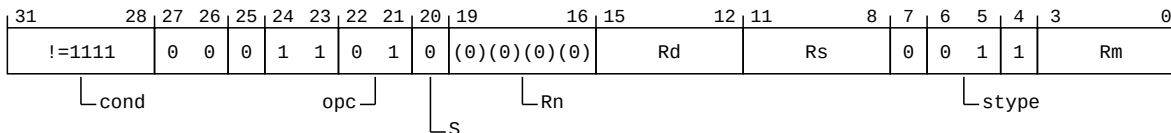
Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Not flag setting variant

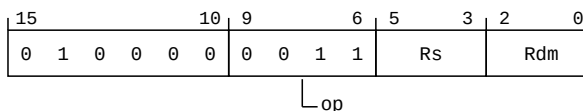
LSR{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

T1



Encoding for the Logical shift right variant

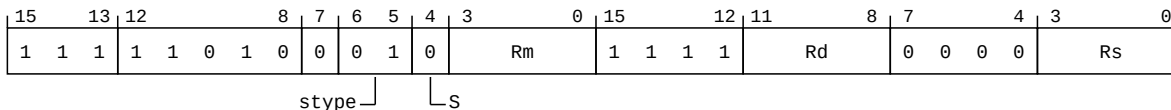
LSR<c>{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>

and is the preferred disassembly when `InITBlock()`.

T2



Encoding for the Not flag setting variant

```
LSR{<c>}{<q>}{<Rd>, }<Rm>, <Rs>
```

```
LSR<c>.W {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOV{<c>}{<q>}{<Rd>, <Rm>, LSR <Rs>}
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>

Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

C5.1.104 LSRS (immediate)

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

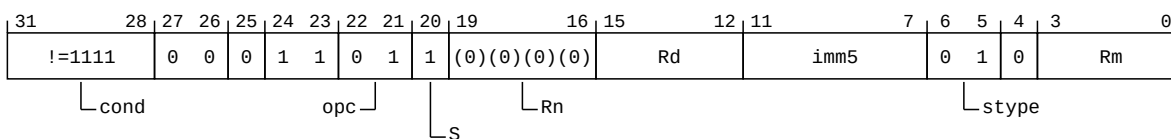
- The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#) and [T3](#)).

A1



Encoding for the MOVS, shift or rotate by value variant

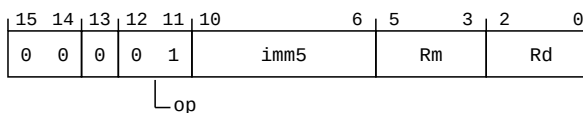
LSRS{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>

and is always the preferred disassembly.

T2



Encoding for the T2 variant

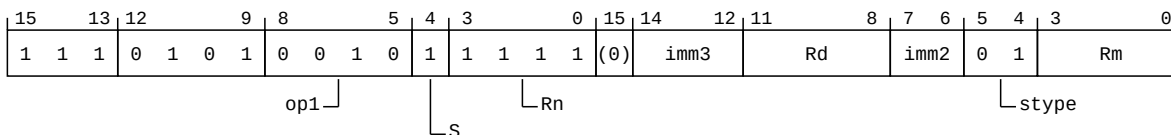
LSRS{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS<c>{<q>} <Rd>, <Rm>, LSR #<imm>

and is the preferred disassembly when !InITBlock().

T3



Encoding for the MOVS, shift or rotate by value variant

```
LSRS{<c>}{<q>} {<Rd>, }<Rm>, #<imm>
```

```
LSRS.W {<Rd>, }<Rm>, #<imm> // (!InITBlock() && UInt(Rd) < 8 && UInt(Rm) < 8)
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T2” and “T3 MOVS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOVS, shift or rotate by value” and “T2” variants: is the shift amount, in the range 1 to 32, encoded in the “imm5” field as <imm> modulo 32.

For the “T3 MOVS, shift or rotate by value” variant: is the shift amount, in the range 1 to 32, encoded in the “imm3:imm2” field as <imm> modulo 32.

Operation

The description of [MOV](#), [MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.105 LSRS (register)

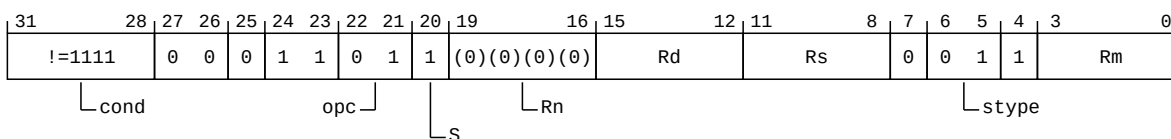
Logical Shift Right, setting flags (register) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Flag setting variant

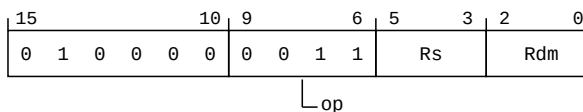
LSRS{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>

and is always the preferred disassembly.

T1



Encoding for the Logical shift right variant

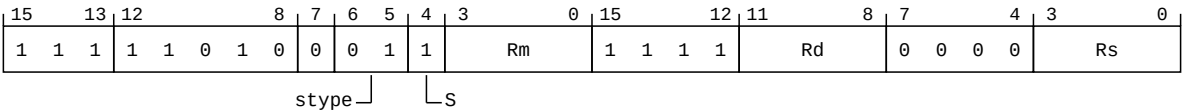
LSRS{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>

and is the preferred disassembly when !InITBlock().

T2



Encoding for the Flag setting variant

```
LSRS{<c>}{<q>} {<Rd>, }<Rm>, <Rs>  
  
LSRS.W {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>
See [Standard assembler syntax fields](#).

<q>
See [Standard assembler syntax fields](#).

<Rd>
Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>
Is the first general-purpose source register, encoded in the “Rm” field.

<Rs>
Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>
Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

C5.1.106 MCR

Move to System register from general-purpose register or execute a System instruction. This instruction copies the value of a general-purpose register to a System register, or executes a System instruction.

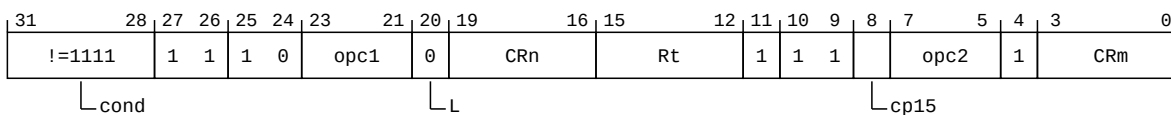
The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MCR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



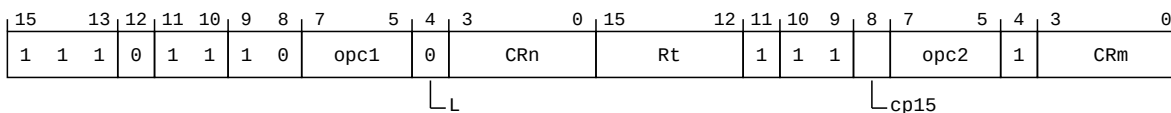
Encoding

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc1” field.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<CRn>

Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRn” field.

<CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRm” field.

<opc2>

Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc2” field.

The possible values of { <coproc>, <opc1>, <CRn>, <CRm>, <opc2> } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegWrite(cp, ThisInstr(), t);
end;
```

C5.1.107 MCRR

Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register.

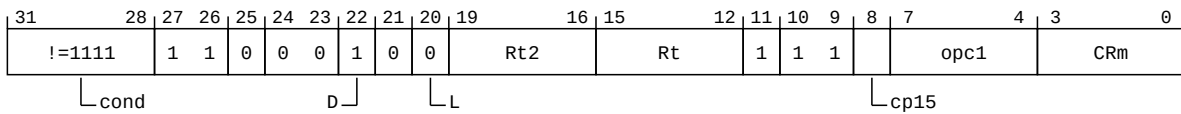
The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MCRR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCRR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MCRR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



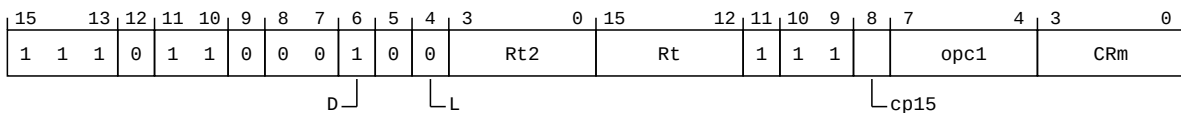
Encoding

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



Encoding

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the “opc1” field.

<Rt>

Is the first general-purpose register that is transferred into, encoded in the “Rt” field.

<Rt2>

Is the second general-purpose register that is transferred into, encoded in the “Rt2” field.

<CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRm” field.

The possible values of { <coproc>, <opc1>, <CRm> } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegWrite64(cp, ThisInstr(), t, t2);
end;
```

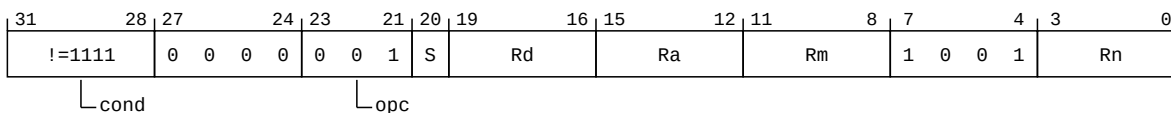
C5.1.108 MLA, MLAS

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

In an A32 instruction, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Flag setting variant

Applies when (S == 1)

MLAS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the Not flag setting variant

Applies when (S == 0)

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

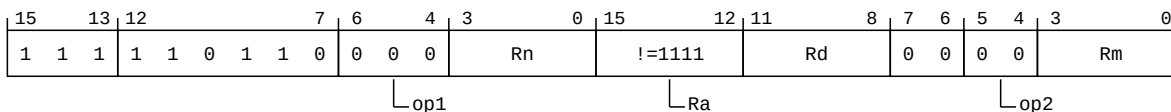
Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let setflags : boolean = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```

if Ra == '1111' then See("MUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let setflags : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let addend : integer = SInt(R(a)); // addend = UInt(R[a]) produces the same final results
    let result : integer = operand1 * operand2 + addend;
    R(d) = result[31:0];
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result[31:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;

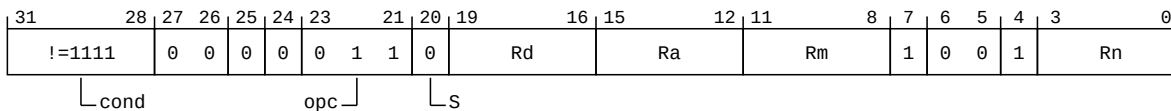
```

C5.1.109 MLS

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



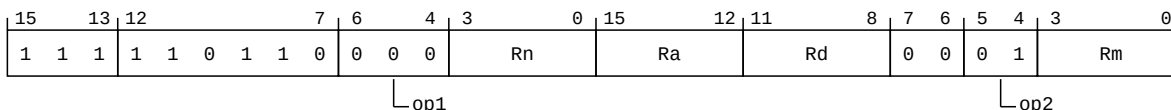
Encoding

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let addend : integer = SInt(R(a)); // addend = UInt(R[a]) produces the same final results
    let result : integer = addend - operand1 * operand2;
    R(d) = result[31:0];
end;
```

C5.1.110 MOV, MOVS (immediate)

Move (immediate) writes an immediate value to the destination register.

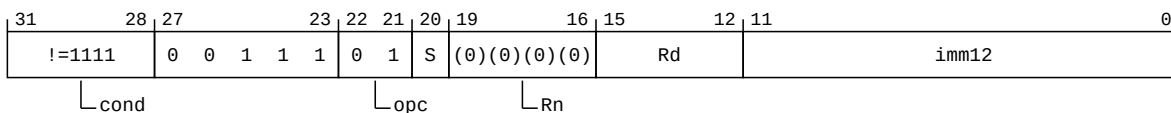
If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MOV variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the MOV variant

Applies when (S == 0)

MOV{<c>}{<q>} <Rd>, #<const>

Encoding for the MOVS variant

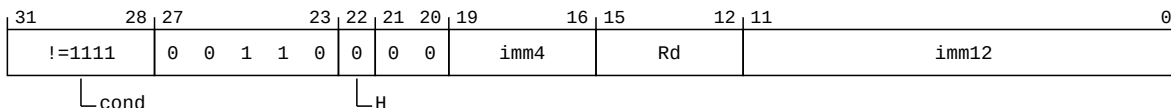
Applies when (S == 1)

MOVS{<c>}{<q>} <Rd>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let (imm32, carry) : (bits(32), bit) = A32ExpandImm_C(imm12, PSTATE.C);
```

A2



Encoding

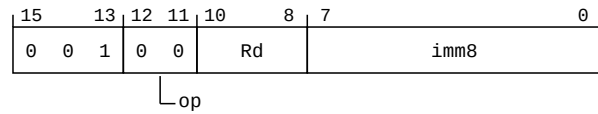
MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can not be represented in A1)

MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in A1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4::imm12);
let carry : bit = ARBITRARY : bit;
if d == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

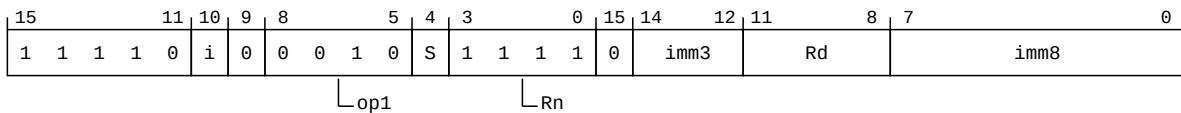
```
MOV<c>{<q>} <Rd>, #<imm8> // (InITBlock())
```

```
MOVS{<q>} <Rd>, #<imm8> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm8);
let carry : bit = PSTATE.C;
```

T2



Encoding for the MOV variant

Applies when (S == 0)

```
MOV{<c>}{<q>} <Rd>, #<const>
```

```
MOV<c>.W <Rd>, #<const> // (Inside IT block, and <Rd>, <const> can be represented in T1)
```

Encoding for the MOVS variant

Applies when (S == 1)

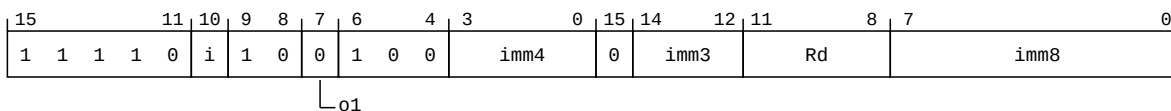
```
MOVS{<c>}{<q>} <Rd>, #<const>
```

```
MOVS.W <Rd>, #<const> // (Outside IT block, and <Rd>, <const> can be represented in T1)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let (imm32, carry) : (bits(32), bit) = T32ExpandImm_C(i::imm3::imm8, PSTATE.C);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

T3



Encoding

```
MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> cannot be represented in T1 or T2)
```

```
MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in T1 or T2)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4::i::imm3::imm8);
let carry : bit = ARBITRARY : bit;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOV” and “A1 MOVS” variants: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the MOVS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “A2”, “T1”, “T2 MOV”, “T2 MOVS”, and “T3” variants: is the general-purpose destination register, encoded in the “Rd” field.

<const>

For the “A1 MOV” and “A1 MOVS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T2 MOV” and “T2 MOVS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

<imm16>

For the “A2” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:imm12” field.

For the “T3” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:i:imm3:imm8” field.

<imm8>

Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : bits(32) = imm32;
    if d == 15 then          // Can only occur for encoding A1
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

C5.1.111 MOV, MOVS (register)

Move (register) copies a value from a register to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

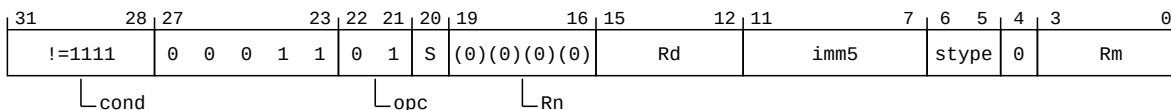
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases [ASRS \(immediate\)](#), [ASR \(immediate\)](#), [LSLS \(immediate\)](#), [LSL \(immediate\)](#), [LSRS \(immediate\)](#), [LSR \(immediate\)](#), [RORS \(immediate\)](#), [ROR \(immediate\)](#), [RRXS](#), and [RRX](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the MOV, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MOV, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

Encoding for the MOVS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

MOVS{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MOVS, shift or rotate by value variant

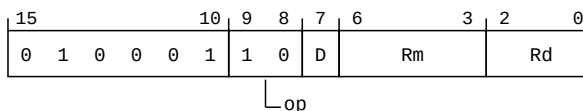
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1



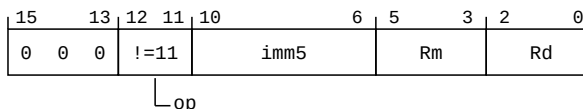
Encoding

MOV{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(D::Rd);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if d == 15 && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2



Encoding

MOV<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (InITBlock())

MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block)

Decode for this encoding

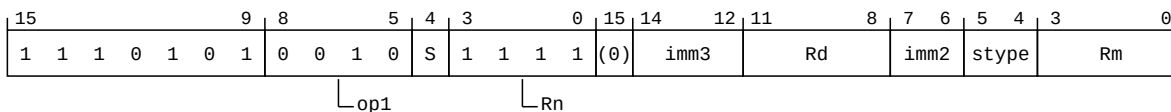
```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(op, imm5);
if op == '00' && imm5 == '00000' && InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If op == '00' && imm5 == '00000' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passed its condition code check.
- The instruction executes as NOP, as if it failed its condition code check.
- The instruction executes as MOV Rd, Rm.

T3



Encoding for the MOV, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

Encoding for the MOV, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

```
MOV{<c>}.W <Rd>, <Rm> {, LSL #0} // (<Rd>, <Rm> can be represented in T1)
```

```
MOV<c>.W <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be  

↪ represented in T2)
```

Encoding for the MOVS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

Encoding for the MOVS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

```
MOVS.W <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be  

↪ represented in T1 or T2)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOV, rotate right with extend”, “A1 MOV, shift or rotate by value”, “A1 MOVS, rotate right with extend”, and “A1 MOVS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). Arm deprecates use of the instruction if <Rn> is the PC.
- For the MOVS variant, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>. Arm deprecates use of the instruction if <Rn> is not the LR, or if the optional shift or RRX argument is specified.

For the “T1” variant: is the general-purpose destination register, encoded in the “D:Rd” field. If the PC is used:

- The instruction causes a branch to the address moved to the PC. This is a simple branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The instruction must either be outside an IT block or the last instruction of an IT block.

For the “T2”, “T3 MOV, rotate right with extend”, “T3 MOV, shift or rotate by value”, “T3 MOVS, rotate right with extend”, and “T3 MOVS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, rotate right with extend”, “A1 MOV, shift or rotate by value”, “A1 MOVS, rotate right with extend”, “A1 MOVS, shift or rotate by value”, and “T1” variants: is the general-purpose source register, encoded in the “Rm” field. The PC can be used. Arm deprecates use of the instruction if <Rd> is the PC.

For the “T2”, “T3 MOV, rotate right with extend”, “T3 MOV, shift or rotate by value”, “T3 MOVS, rotate right with extend”, and “T3 MOVS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<shift>

For the “A1 MOV, shift or rotate by value”, “A1 MOVS, shift or rotate by value”, “T3 MOV, shift or rotate by value”, and “T3 MOVS, shift or rotate by value” variants: is the type of shift to be applied to the source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

For the “T2” variant: is the type of shift to be applied to the source register, encoded in “op”:

op	<shift>
00	LSL
01	LSR
10	ASR

<amount>

For the “A1 MOV, shift or rotate by value” and “A1 MOVs, shift or rotate by value” variants: is the shift amount, in the range 0 to 31 (when <shift> = LSL), or 1 to 31 (when <shift> = ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T2” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T3 MOV, shift or rotate by value” and “T3 MOVs, shift or rotate by value” variants: is the shift amount, in the range 0 to 31 (when <shift> = LSL) or 1 to 31 (when <shift> = ROR), or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Alias Conditions

Alias	Of variant	Is preferred when
ASRS (immediate)	T3 MOVs, shift or rotate by value, A1 MOVs, shift or rotate by value	S == '1' && stype == '10'
ASRS (immediate)	T2	op == '10' && !InITBlock()
ASRS (immediate)	T3 MOVs, shift or rotate by value	Unconditionally
ASR (immediate)	T3 MOV, shift or rotate by value, A1 MOV, shift or rotate by value	S == '0' && stype == '10'
ASR (immediate)	T2	op == '10' && InITBlock()
ASR (immediate)	T3 MOV, shift or rotate by value	Unconditionally
LSLS (immediate)	A1 MOVs, shift or rotate by value	S == '1' && imm5 != '00000' && stype == '00'
LSLS (immediate)	T2	op == '00' && imm5 != '00000' && !InITBlock()
LSLS (immediate)	T3 MOVs, shift or rotate by value	(S == '1' && stype == '00' && !(imm3 == '000')) (S == '1' && stype == '00' && !(imm2 == '00'))
LSLS (immediate)	T3 MOVs, shift or rotate by value	Unconditionally
LSL (immediate)	A1 MOV, shift or rotate by value	S == '0' && imm5 != '00000' && stype == '00'
LSL (immediate)	T2	op == '00' && imm5 != '00000' && InITBlock()
LSL (immediate)	T3 MOV, shift or rotate by value	(S == '0' && stype == '00' && !(imm3 == '000')) (S == '0' && stype == '00' && !(imm2 == '00'))

Alias	Of variant	Is preferred when
LSL (immediate)	T3 MOV, shift or rotate by value	Unconditionally
LSRS (immediate)	T3 MOVS, shift or rotate by value, A1 MOVS, shift or rotate by value	S == '1' && stype == '01'
LSRS (immediate)	T2	op == '01' && !InITBlock()
LSRS (immediate)	T3 MOVS, shift or rotate by value	Unconditionally
LSR (immediate)	T3 MOV, shift or rotate by value, A1 MOV, shift or rotate by value	S == '0' && stype == '01'
LSR (immediate)	T2	op == '01' && InITBlock()
LSR (immediate)	T3 MOV, shift or rotate by value	Unconditionally
RORS (immediate)	A1 MOVS, shift or rotate by value	S == '1' && imm5 != '00000' && stype == '11'
RORS (immediate)	T3 MOVS, shift or rotate by value	(S == '1' && stype == '11' && !(imm3 == '000')) (S == '1' && stype == '11' && !(imm2 == '00'))
ROR (immediate)	A1 MOV, shift or rotate by value	S == '0' && imm5 != '00000' && stype == '11'
ROR (immediate)	T3 MOV, shift or rotate by value	(S == '0' && stype == '11' && !(imm3 == '000')) (S == '0' && stype == '11' && !(imm2 == '00'))
RRXS	A1 MOVS, rotate right with extend	S == '1' && imm5 == '00000' && stype == '11'
RRXS	T3 MOVS, rotate right with extend	S == '1' && imm3 == '000' && imm2 == '00' && stype == '11'
RRX	A1 MOV, rotate right with extend	S == '0' && imm5 == '00000' && stype == '11'
RRX	T3 MOV, rotate right with extend	S == '0' && imm3 == '000' && imm2 == '00' && stype == '11'

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = shifted;
    if d == 15 then
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

```

```
        end;  
    end;  
end;
```

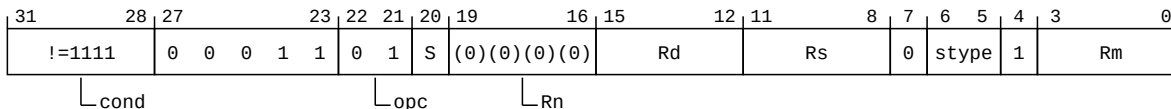
C5.1.112 MOV, MOVS (register-shifted register)

Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases [ASRS \(register\)](#), [ASR \(register\)](#), [LSLS \(register\)](#), [LSL \(register\)](#), [LSRS \(register\)](#), [LSR \(register\)](#), [RORS \(register\)](#), and [ROR \(register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Flag setting variant

Applies when (S == 1)

```
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

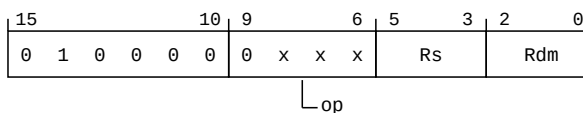
Applies when (S == 0)

```
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the Arithmetic shift right variant

Applies when (op == 0100)

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs> // (InITBlock())
```

```
MOV{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Outside IT block)
```

Encoding for the Logical shift left variant

Applies when (op == 0010)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs> // (InITBlock())

MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Outside IT block)
```

Encoding for the Logical shift right variant

Applies when (op == 0011)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs> // (InITBlock())

MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Outside IT block)
```

Encoding for the Rotate right variant

Applies when (op == 0111)

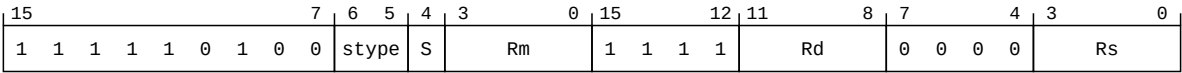
```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs> // (InITBlock())

MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs> // (Outside IT block)
```

Decode for all variants of this encoding

```
if ! op IN {'0010', '0011', '0100', '0111'} then See("Related encodings"); end;
let d : integer = UInt(Rdm);
let m : integer = UInt(Rdm);
let s : integer = UInt(Rs);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = DecodeRegShift(op[2]:op[0]);
```

T2



Encoding for the Flag setting variant

Applies when (S == 1)

```
MOV<c>{<q>} <Rd>, <Rm>, <shift> <Rs>

MOV<c>.W <Rd>, <Rm>, <shift> <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented
↳in T1)
```

Encoding for the Not flag setting variant

Applies when (S == 0)

```
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

MOV<c>.W <Rd>, <Rm>, <shift> <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented
↳in T1)
```


Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

Related encodings: In encoding T1, for an `op` field value that is not described above, see [Data-processing \(two low registers\)](#).

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the general-purpose source register and the destination register, encoded in the “Rdm” field.

Alias Conditions

Alias	Of variant	Is preferred when
ASRS (register)	A1 Flag setting	S == '1' && stype == '10'
ASRS (register)	T1 Arithmetic shift right	op == '0100' && !InITBlock()
ASRS (register)	T2 Flag setting	stype == '10' && S == '1'
ASRS (register)	T2 Flag setting	Unconditionally
ASR (register)	A1 Not flag setting	S == '0' && stype == '10'
ASR (register)	T1 Arithmetic shift right	op == '0100' && InITBlock()
ASR (register)	T2 Not flag setting	stype == '10' && S == '0'
ASR (register)	T2 Not flag setting	Unconditionally
LSLS (register)	A1 Flag setting	S == '1' && stype == '00'
LSLS (register)	T1 Logical shift left	op == '0010' && !InITBlock()
LSLS (register)	T2 Flag setting	stype == '00' && S == '1'
LSLS (register)	T2 Flag setting	Unconditionally
LSL (register)	A1 Not flag setting	S == '0' && stype == '00'
LSL (register)	T1 Logical shift left	op == '0010' && InITBlock()
LSL (register)	T2 Not flag setting	stype == '00' && S == '0'
LSL (register)	T2 Not flag setting	Unconditionally
LSRS (register)	A1 Flag setting	S == '1' && stype == '01'
LSRS (register)	T1 Logical shift right	op == '0011' && !InITBlock()
LSRS (register)	T2 Flag setting	stype == '01' && S == '1'
LSRS (register)	T2 Flag setting	Unconditionally
LSR (register)	A1 Not flag setting	S == '0' && stype == '01'
LSR (register)	T1 Logical shift right	op == '0011' && InITBlock()
LSR (register)	T2 Not flag setting	stype == '01' && S == '0'
LSR (register)	T2 Not flag setting	Unconditionally
RORS (register)	A1 Flag setting	S == '1' && stype == '11'
RORS (register)	T1 Rotate right	op == '0111' && !InITBlock()
RORS (register)	T2 Flag setting	stype == '11' && S == '1'
RORS (register)	T2 Flag setting	Unconditionally
ROR (register)	A1 Not flag setting	S == '0' && stype == '11'
ROR (register)	T1 Rotate right	op == '0111' && InITBlock()
ROR (register)	T2 Not flag setting	stype == '11' && S == '0'
ROR (register)	T2 Not flag setting	Unconditionally

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (result, carry) : (bits(32), bit) = Shift_C[32](R(m), shift_t, shift_n, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit[32](result);

```

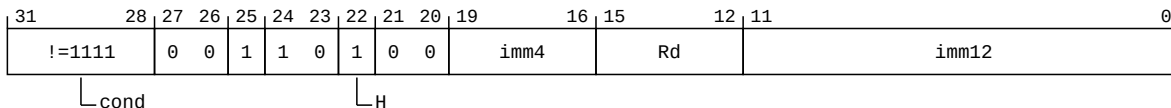
```
        PSTATE.C = carry;  
        // PSTATE.V unchanged  
    end;  
end;
```

C5.1.113 MOV_T

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



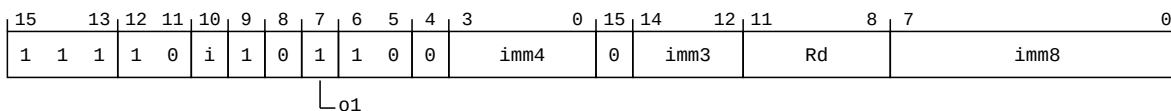
Encoding

MOV_T{<c>}{<q>} <Rd>, #<imm16>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm16 : bits(16) = imm4::imm12;
if d == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

MOV_T{<c>}{<q>} <Rd>, #<imm16>

Decode for this encoding

```
let d : integer = UInt(Rd);
let imm16 : bits(16) = imm4::i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm16>

For the “A1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:imm12” field.

For the “T1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:i:imm3:imm8” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[31:16] = imm16;
    // R[d]<15:0> unchanged
end;
```

C5.1.114 MRC

Move to general-purpose register from System register. This instruction copies the value of a System register to a general-purpose register.

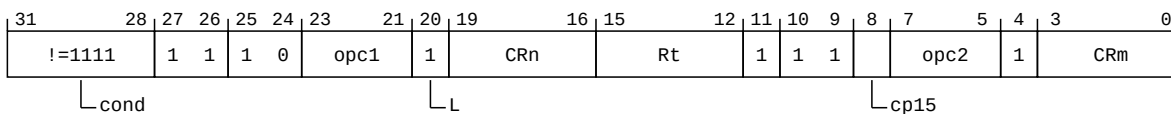
The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MRC accesses to system control registers can be trapped to Hyp mode, meaning that an attempt to execute an MRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



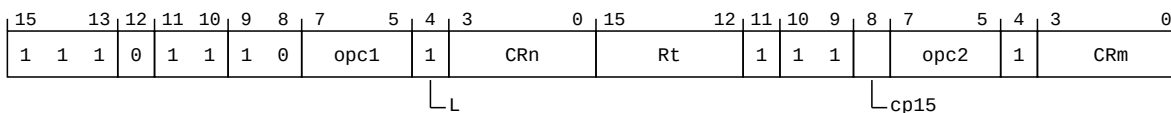
Encoding

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



Encoding

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the opc1 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc1” field.

<Rt>

Is the general-purpose register to be transferred or APSR_nzcv (encoded as 0b1111), encoded in the “Rt” field. If APSR_nzcv is used, bits [31:28] of the transferred value are written to the [PSTATE](#) condition flags.

<CRn>

Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRn” field.

<CRm>

Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRm” field.

<opc2>

Is the opc2 parameter within the System register encoding space, in the range 0 to 7, encoded in the “opc2” field.

The possible values of { <coproc>, <opc1>, <CRn>, <CRm>, <opc2> } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    if t != 15 || AArch32_SysRegReadCanWriteAPSR(cp, ThisInstr()) then
        AArch32_SysRegRead(cp, ThisInstr(), t);
    else
        UnpredictableProcedure();
    end;
end;
```

C5.1.115 MRRC

Move to two general-purpose registers from System register. This instruction copies the value of a System register to two general-purpose registers.

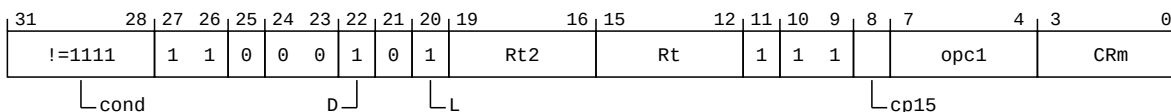
The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MRRC accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MRRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MRRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

Decode for this encoding

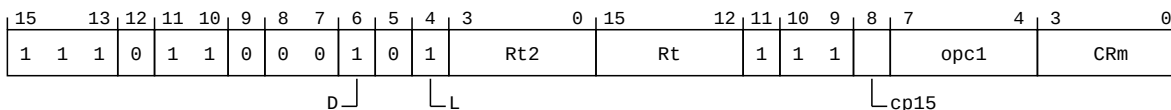
```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $t == t2$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let cp : integer = if cp15 == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<coproc>

Is the System register encoding space, encoded in “cp15”:

cp15	<coproc>
0	p14
1	p15

<opc1>

Is the `opc1` parameter within the System register encoding space, in the range 0 to 15, encoded in the “opc1” field.

<Rt>

Is the first general-purpose register that is transferred into, encoded in the “Rt” field.

<Rt2>

Is the second general-purpose register that is transferred into, encoded in the “Rt2” field.

<CRm>

Is the `CRm` parameter within the System register encoding space, in the range c0 to c15, encoded in the “CRm” field.

The possible values of { <coproc>, <opc1>, <CRm> } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_SysRegRead64(cp, ThisInstr(), t, t2);
end;
```

C5.1.116 MRS

Move Special register to general-purpose register moves the value of the [APSR](#), [CPSR](#), or [SPSR](#) `<current_mode>` into a general-purpose register.

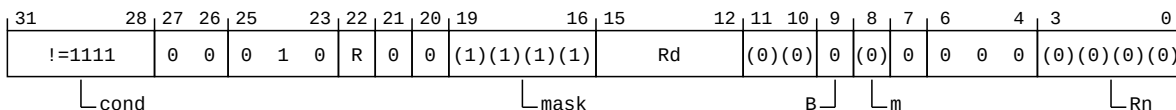
Arm recommends the [APSR](#) form when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see [APSR](#).

An MRS that accesses the [SPSRs](#) is UNPREDICTABLE if executed in User mode or System mode.

An MRS that is executed in User mode and accesses the [CPSR](#) returns an UNKNOWN value for the [CPSR](#). {E, A, I, F, M} fields.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



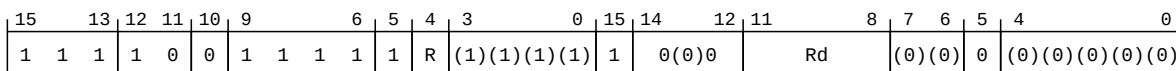
Encoding

MRS{<c>}{<q>} <Rd>, <spec_reg>

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
if d == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

MRS{<c>}{<q>} <Rd>, <spec_reg>

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<spec_reg>

Is the special register to be accessed, encoded in “R”:

R	<spec_reg>
0	CPSR APSR
1	SPSR

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if read_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        else
            R(d) = SPSR_curr();
        end;
    else
        // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
        let maskval : bits(32) = '11111000 11101111 00000011 11011111';
        var psr_val : bits(32) = GetPSRFromPSTATE{32}(AArch32_NonDebugState) AND maskval;
        if PSTATE.EL == EL0 then
            // If accessed from User mode return UNKNOWN values for E, A, I, F bits, bits<9:6>,
            // and for the M field, bits<4:0>
            psr_val[9:6] = ARBITRARY : bits(4);
            psr_val[4:0] = ARBITRARY : bits(5);
        end;
        R(d) = psr_val;
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User, M32_System} && read_spsr, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

C5.1.117 MRS (Banked register)

Move Banked or Special register to general-purpose register

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or [Saved Program Status Registers \(SPSRs\)](#) of the specified mode, or the value of [ELR_hyp](#), to a general-purpose register.

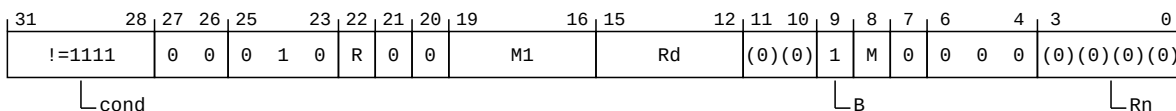
MRS (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MRS (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MRS (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see [Usage restrictions on the Banked register transfer instructions](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



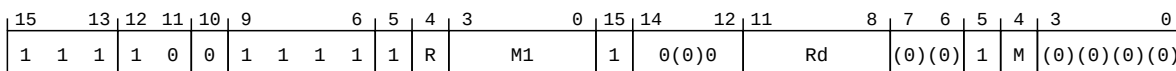
Encoding

MRS{<c>}{<q>} <Rd>, <banked_reg>

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
if d == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

T1



Encoding

MRS{<c>}{<q>} <Rd>, <banked_reg>

Decode for this encoding

```
let d : integer = UInt(Rd);
let read_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<banked_reg>

Is the name of the banked register to be transferred to or from, encoded in “R:M:M1”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon

R	M	M1	<banked_reg>
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == ELO then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_BankedRegister);
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP =>
                ExecuteAsNOP();
            when Constraint_UNKNOWN =>
                R(d) = ARBITRARY : bits(32);
        end;
    else
        let mode : bits(5) = PSTATE.M;
        if read_spsr then
            // Check for CONSTRAINED UNPREDICTABLE cases
            var valid : boolean;
            var c : Constraint = Constraint_NONE;
            (valid, c) = SPSRaccessValid(SYSm, mode);

            if valid then
                case SYSm of
                    when '01110' => R(d) = SPSR_fiq()[31:0];
                    when '10000' => R(d) = SPSR_irq()[31:0];
                    when '10010' => R(d) = SPSR_svc()[31:0];
                    when '10100' => R(d) = SPSR_abt()[31:0];
                    when '10110' => R(d) = SPSR_und()[31:0];
                    when '11110' => R(d) = SPSR_hyp()[31:0];
                end;
            end;
        end;
    end;

```

```

else
    case c of
        when Constraint_UNDEF =>
            Undefined();
        when Constraint_NOP =>
            ExecuteAsNOP();
        when Constraint_UNKNOWN =>
            R(d) = ARBITRARY : bits(32);
        when Constraint_ANYREG =>
            R(d) = ReadAnyAllocatedSPSR();
    end;
end;
else
    // Check for CONSTRAINED UNPREDICTABLE cases
    var valid : boolean;
    var c : Constraint = Constraint_NONE;
    (valid, c) = BankedRegisterAccessValid(SYSm, mode);

    if valid then
        case SYSm of
            when '00xxx' => // Access the User mode registers
                let m : integer = UInt(SYSm[2:0]) + 8;
                R(d) = Rmode(m, M32_User);
            when '01xxx' => // Access the FIQ mode registers
                let m : integer = UInt(SYSm[2:0]) + 8;
                R(d) = Rmode(m, M32_FIQ);
            when '1000x' => // Access the IRQ mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m, M32_IRQ);
            when '1001x' => // Access the Supervisor mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m, M32_Svc);
            when '1010x' => // Access the Abort mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m, M32_Abort);
            when '1011x' => // Access the Undefined mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m, M32_Undef);
            when '1110x' => // Access Monitor registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                R(d) = Rmode(m, M32_Monitor);
            when '11110' => // Access ELR_hyp register
                R(d) = ELR_hyp();
            when '11111' => // Access SP_hyp register
                R(d) = Rmode(13, M32_Hyp);
        end;
    else
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP =>
                ExecuteAsNOP();
            when Constraint_UNKNOWN =>
                R(d) = ARBITRARY :bits(32);
            when Constraint_ANYREG =>
                R(d) = ReadAnyAllocatedRegister();
        end;
    end;
end;
end;
end;
end;
end;

```


CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.

C5.1.118 MSR (Banked register)

Move general-purpose register to Banked or Special register

Move to Banked or Special register from general-purpose register moves the value of a general-purpose register to the Banked general-purpose register or [Saved Program Status Registers \(SPSRs\)](#) of the specified mode, or to [ELR_hyp](#).

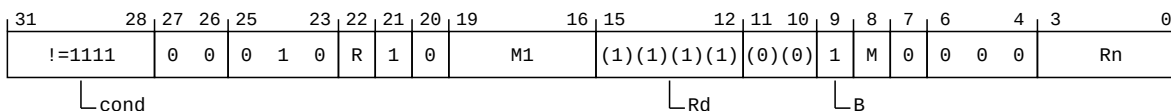
MSR (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MSR (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MSR (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see [Usage restrictions on the Banked register transfer instructions](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



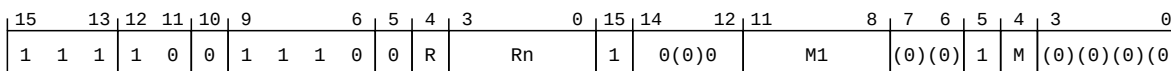
Encoding

MSR{<c>}{<q>} <banked_reg>, <Rn>

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if n == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

T1



Encoding

MSR{<c>}{<q>} <banked_reg>, <Rn>

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
let SYSm : bits(5) = M::M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<banked_reg>

Is the name of the banked register to be transferred to or from, encoded in “R:M:M1”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE

R	M	M1	<banked_reg>
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == ELO then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_BankedRegister);
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP, Constraint_UNKNOWN =>
                ExecuteAsNOP();
        end;
    else
        let mode : bits(5) = PSTATE.M;
        if write_spsr then
            // Check for CONSTRAINED UNPREDICTABLE cases
            var valid : boolean;
            var c : Constraint = Constraint_NONE;
            (valid, c) = SPSRaccessValid(SYSm, mode);

            if valid then
                case SYSm of
                    when '01110' => SPSR_fiq() [31:0] = R(n);
                    when '10000' => SPSR_irq() [31:0] = R(n);
                    when '10010' => SPSR_svc() [31:0] = R(n);
                    when '10100' => SPSR_abt() [31:0] = R(n);
                    when '10110' => SPSR_und() [31:0] = R(n);
                    when '11110' => SPSR_hyp() [31:0] = R(n);
                end;
            else
                case c of
                    when Constraint_UNDEF =>

```

```

        Undefined();
    when Constraint_NOP, Constraint_UNKNOWN =>
        ExecuteAsNOP();
    when Constraint_ANYREG =>
        WriteAnyAllocatedSPSR(R(n));
    end;
end;
else
    // Check for CONSTRAINED UNPREDICTABLE cases
    var valid : boolean;
    var c : Constraint = Constraint_NONE;
    (valid, c) = BankedRegisterAccessValid(SYSm, mode);

    if valid then
        case SYSm of
            when '00xxx' => // Access the User mode registers
                let m : integer = UInt(SYSm[2:0]) + 8;
                Rmode(m, M32_User) = R(n);
            when '01xxx' => // Access the FIQ mode registers
                let m : integer = UInt(SYSm[2:0]) + 8;
                Rmode(m, M32_FIQ) = R(n);
            when '1000x' => // Access the IRQ mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                Rmode(m, M32_IRQ) = R(n);
            when '1001x' => // Access the Supervisor mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                Rmode(m, M32_Svc) = R(n);
            when '1010x' => // Access the Abort mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                Rmode(m, M32_Abort) = R(n);
            when '1011x' => // Access the Undefined mode registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                Rmode(m, M32_Undef) = R(n);
            when '1110x' => // Access Monitor registers
                // LR when SYSm[0] == 0, otherwise SP
                let m : integer = 14 - UInt(SYSm[0]);
                Rmode(m, M32_Monitor) = R(n);
            when '11110' => // Access ELR_hyp register
                ELR_hyp() = R(n);
            when '11111' => // Access SP_hyp register
                Rmode(13, M32_Hyp) = R(n);
        end;
    else
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP, Constraint_UNKNOWN =>
                ExecuteAsNOP();
            when Constraint_ANYREG =>
                WriteAnyAllocatedRegister(R(n));
        end;
    end;
end;
end;
end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as `NOP`.

C5.1.119 MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the [APSR](#), [CPSR](#), or [SPSR](#)_*<current_mode>*.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of `MSR` is normally only useful at the Application level for writing to `APSR_nzcvq` (`CPSR_ℱ`).

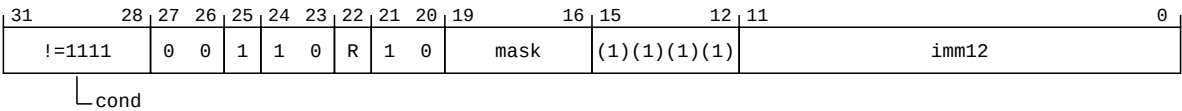
If an `MSR` (immediate) moves selected bits of an immediate value to the [CPSR](#), the PE checks whether the value being written to `PSTATE.M` is legal. See [Illegal changes to PSTATE.M](#).

An `MSR` (immediate) executed in User mode:

- Is **CONSTRAINED UNPREDICTABLE** if it attempts to update the [SPSR](#).
- Otherwise, does not update any [CPSR](#) field that is accessible only at EL1 or higher,

An `MSR` (immediate) executed in System mode is **CONSTRAINED UNPREDICTABLE** if it attempts to update the [SPSR](#).

The [CPSR.E](#) bit is writable from any mode using an `MSR` instruction. Arm deprecates using this to change its value.



Encoding

Applies when `(!(R == 0 && mask == 0000))`

```
MSR{<c>}{<q>} <spec_reg>, #<imm>
```

Decode for this encoding

```
if mask == '0000' && R == '0' then See("Related encodings"); end;
let imm32 : bits(32) = A32ExpandImm(imm12);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000' && R == '1'`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Move Special Register and Hints \(immediate\)](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<spec_reg>

Is one of:

- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

- c** mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.
- x** mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.
- s** mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.
- f** mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvq, g, or nzcvqg. These map to the following CPSR_<fields> values:

- APSR_nzcvq is the same as CPSR_f (mask == '1000').
- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvqg is the same as CPSR_fs (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see [The Application Program Status Register, APSR](#).

<imm>

Is an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        else
            SPSRWriteByInstr(imm32, mask);
        end;
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(imm32, mask);
    end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User,M32_System} && write_spsr, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

C5.1.120 MSR (register)

Move general-purpose register to Special register moves selected bits of a general-purpose register to the [APSR](#), [CPSR](#) or [SPSR](#)_*<current_mode>*.

Because of the Do-Not-Modify nature of its reserved bits, a read-modify-write sequence is normally required when the MSR instruction is being used at Application level and its destination is not [APSR_nzcvq](#) ([CPSR_f](#)).

If an MSR (register) moves selected bits of an immediate value to the [CPSR](#), the PE checks whether the value being written to [PSTATE.M](#) is legal. See [Illegal changes to PSTATE.M](#).

An MSR (register) executed in User mode:

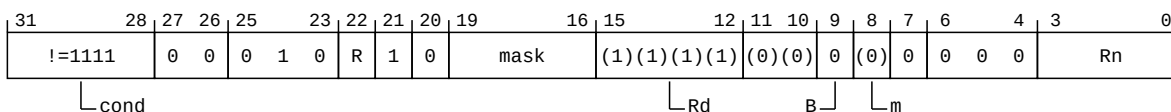
- Is UNPREDICTABLE if it attempts to update the [SPSR](#).
- Otherwise, does not update any [CPSR](#) field that is accessible only at EL1 or higher.

An MSR (register) executed in System mode is UNPREDICTABLE if it attempts to update the [SPSR](#).

The [CPSR.E](#) bit is writable from any mode using an MSR instruction. Arm deprecates using this to change its value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

MSR{<c>}{<q>} <spec_reg>, <Rn>

Decode for this encoding

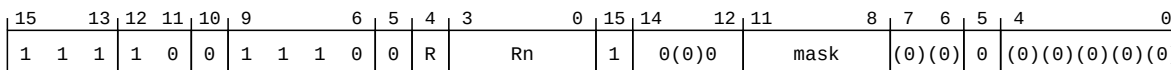
```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
if n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If mask == '0000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

T1



Encoding

MSR{<c>}{<q>} <spec_reg>, <Rn>

Decode for this encoding

```
let n : integer = UInt(Rn);
let write_spsr : boolean = (R == '1');
if mask == '0000' then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<spec_reg>

Is one of:

- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.
x mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.
s mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.
f mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of `nzcvq`, `g`, or `nzcvqg`. These map to the following CPSR_<fields> values:

- APSR_<nzcvq> is the same as CPSR_<f> (mask == '1000').
- APSR_<g> is the same as CPSR_<s> (mask == '0100').
- APSR_<nzcvqg> is the same as CPSR_<fs> (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see [The Application Program Status Register, APSR](#).

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User, M32_System} then
            UnpredictableProcedure();
        else
            SPSRWriteByInstr(R(n), mask);
        end;
```

```
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(R(n), mask);
    end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `write_spsr && PSTATE.M` IN {M32_User, M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

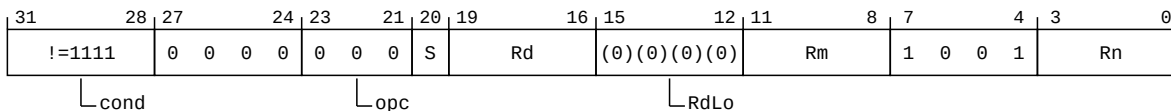
C5.1.121 MUL, Muls

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

Optionally, it can update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1



Encoding for the Flag setting variant

Applies when (S == 1)

MULS{<c>}{<q>} <Rd>, <Rn>{, <Rm>}

Encoding for the Not flag setting variant

Applies when (S == 0)

MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}

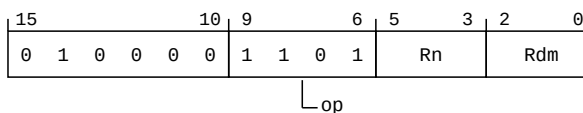
Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // (InITBlock())

MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // (Outside IT block)

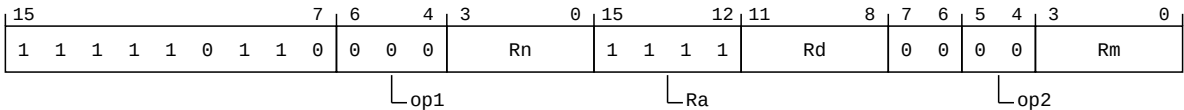
Decode for this encoding

```

let d : integer = UInt(Rdm);
let n : integer = UInt(Rn);
let m : integer = UInt(Rdm);
let setflags : boolean = !InITBlock();

```

T2



Encoding

```
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}  
  
MUL<c>.W <Rd>, <Rn>{, <Rm>} // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);  
let n : integer = UInt(Rn);  
let m : integer = UInt(Rm);  
let setflags : boolean = FALSE;  
// Armv8-A removes UNPREDICTABLE for R13  
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

- <c>
See [Standard assembler syntax fields](#).
- <q>
See [Standard assembler syntax fields](#).
- <Rd>
Is the general-purpose destination register, encoded in the “Rd” field.
- <Rn>
Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.
- <Rm>
Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field. If omitted, <Rd> is used.
- <Rdm>
Is the second general-purpose source register holding the multiplier and the destination register, encoded in the “Rdm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : integer = SInt(R(n)); // operand1 = UInt(R[n]) produces the same final results
    let operand2 : integer = SInt(R(m)); // operand2 = UInt(R[m]) produces the same final results
    let result : integer = operand1 * operand2;
    R(d) = result[31:0];
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result[31:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

C5.1.122 MVN, MVNS (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.

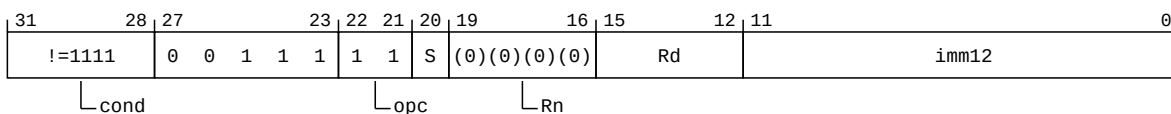
If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the MVN variant

Applies when (S == 0)

MVN{<c>}{<q>} <Rd>, #<const>

Encoding for the MVNS variant

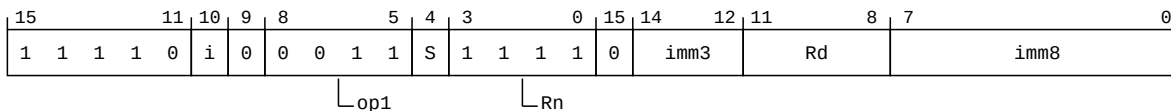
Applies when (S == 1)

MVNS{<c>}{<q>} <Rd>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding for the MVN variant

Applies when (S == 0)

MVN{<c>}{<q>} <Rd>, #<const>

Encoding for the MVNS variant

Applies when (S == 1)

MVNS{<c>}{<q>} <Rd>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MVN” and “A1 MVNS” variants: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the MVNS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T1 MVN” and “T1 MVNS” variants: is the general-purpose destination register, encoded in the “Rd” field.

<const>

For the “A1 MVN” and “A1 MVNS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1 MVN” and “T1 MVNS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                           else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = NOT(imm32);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    end;
```

```
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

C5.1.123 MVN, MVNS (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register.

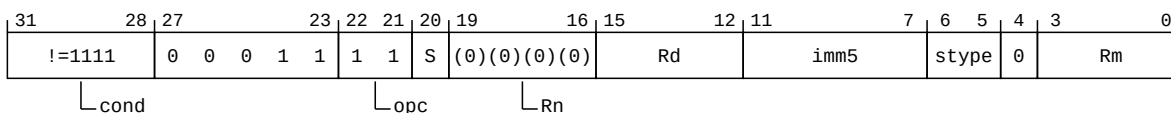
If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the MVN, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

MVN{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MVN, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

Encoding for the MVNS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

MVNS{<c>}{<q>} <Rd>, <Rm>, RRX

Encoding for the MVNS, shift or rotate by value variant

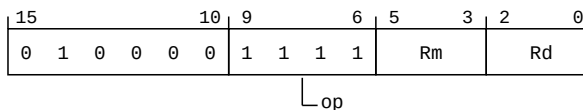
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1



Encoding

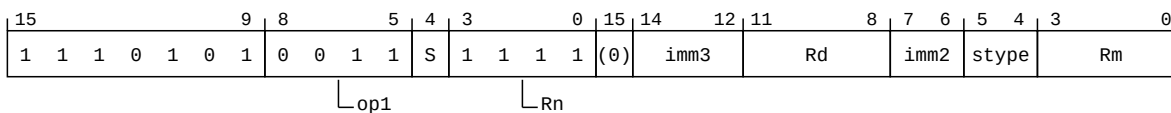
```
MVN<c>{<q>} <Rd>, <Rm> // (InITBlock())
```

```
MVNS{<q>} <Rd>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the MVN, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

Encoding for the MVN, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

```
MVN<c>.W <Rd>, <Rm> // (Inside IT block, and <Rd>, <Rm> can be represented in T1)
```

Encoding for the MVNS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

Encoding for the MVNS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

```
MVNS.W <Rd>, <Rm> // (Outside IT block, and <Rd>, <Rm> can be represented in T1)
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MVN, rotate right with extend”, “A1 MVN, shift or rotate by value”, “A1 MVNS, rotate right with extend”, and “A1 MVNS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the MVNS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T1”, “T2 MVN, rotate right with extend”, “T2 MVN, shift or rotate by value”, “T2 MVNS, rotate right with extend”, and “T2 MVNS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MVN, rotate right with extend”, “A1 MVN, shift or rotate by value”, “A1 MVNS, rotate right with extend”, and “A1 MVNS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 MVN, rotate right with extend”, “T2 MVN, shift or rotate by value”, “T2 MVNS, rotate right with extend”, and “T2 MVNS, shift or rotate by value” variants: is the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 MVN, shift or rotate by value” and “A1 MVNS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

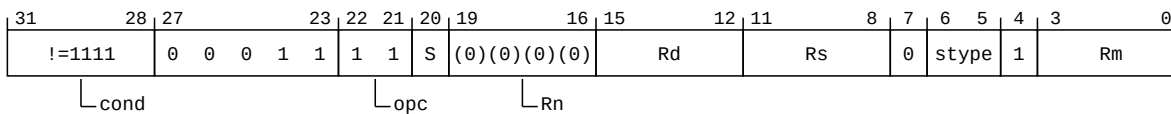
For the “T2 MVN, shift or rotate by value” and “T2 MVNS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;
```

C5.1.124 MVN, MVNS (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

MVNS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

MVN{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SType = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = NOT(shifted);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

C5.1.125 NOP

No Operation does nothing. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0				
!=1111				0 0		1	1 0		0	1 0		0 0 0 0		(1)(1)(1)(1)		(0)(0)(0)(0)00000000			
└─ cond				└─ R				└─ imm4				└─ imm12							

Encoding

NOP{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1

15	12	11	8	7	4	3	0
1	0	1	1	1	1	1	1
└─ hint							

Encoding

NOP{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2

15	13	12	11	10	9	6	5	4	3	0	15	14	12	11	10	8	7	4	3	0	
1	1	1	1	0	0	1	1	1	0	1	0	(1)(1)(1)(1)	1	0(0)0	(0)	0	0	0	0	0	0
															hint			option			

Encoding

NOP{<c>}.w

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

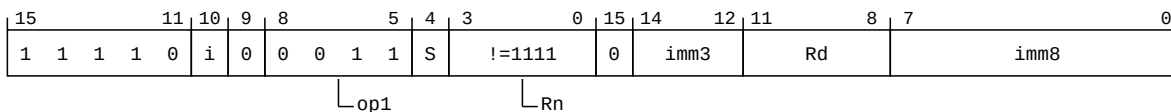
See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
end;
```

C5.1.126 ORN, ORNS (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

ORNS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the Not flag setting variant

Applies when (S == 0)

ORN{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```

if Rn == '1111' then See("MVN (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<const>

An immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

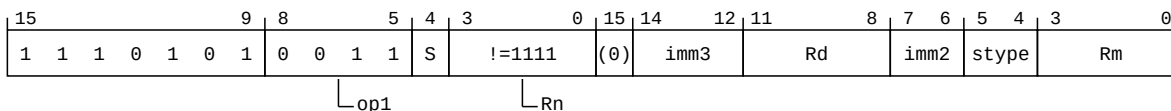
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                             else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) OR NOT(imm32);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

C5.1.127 ORN, ORNS (register)

Bitwise OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.



Encoding for the ORN, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

ORN{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ORN, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ORN{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the ORNS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

ORNS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ORNS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

ORNS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```

if Rn == '1111' then See("MVN (register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR NOT(shifted);
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

C5.1.128 ORR, ORRS (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

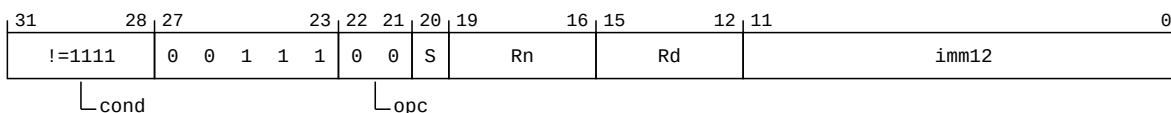
If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the ORR variant

Applies when (S == 0)

```
ORR{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Encoding for the ORRS variant

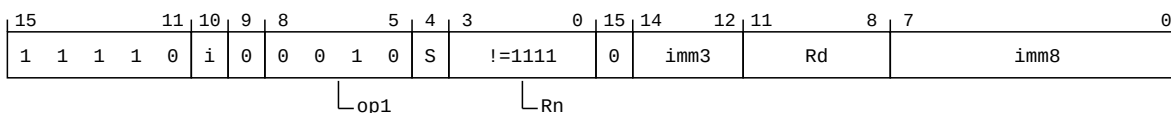
Applies when (S == 1)

```
ORRS{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding for the ORR variant

Applies when (S == 0)

```
ORR{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Encoding for the ORRS variant

Applies when (S == 1)

```
ORRS{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("MOV (immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 ORR” and “A1 ORRS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ORRS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T1 ORR” and “T1 ORRS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 ORR” and “A1 ORRS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 ORR” and “T1 ORRS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 ORR” and “A1 ORRS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1 ORR” and “T1 ORRS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                             else T32ExpandImm_C(imm, PSTATE.C));

    let result : bits(32) = R(n) OR imm32;
    if d == 15 then                // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit{32}(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
```

C5.1.129 ORR, ORRS (register)

Bitwise OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

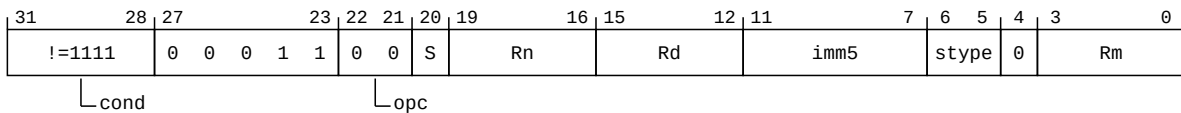
If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the ORR, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

ORR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ORR, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

ORR{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the ORRS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

ORRS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ORRS, shift or rotate by value variant

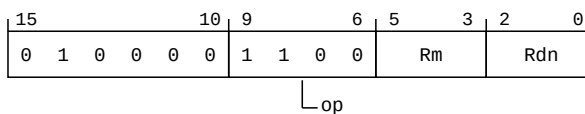
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

ORRS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1



Encoding

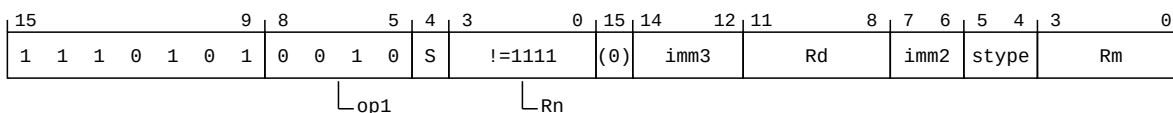
```
ORR<c>{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

ORRS{<g>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift n : integer = 0;
```

T2



Encoding for the ORR, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

ORR { <c> } { <q> } { <Rd> , } <Rn> , <Rm> , RRX

Encoding for the ORR, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ORR{<c>}{<q>}{<Rd>,<Rn>,<Rm>{,<shift>#<amount>}
```

ORR<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the ORRS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

ORRS {<c>} {<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the ORRS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ORRS {<c>} {<q>} {<Rd> , } <Rn> , <Rm> { , <shift> #<amount> }
```

ORRS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
if Rn == '1111' then See("Related encodings"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Data-processing \(shifted register\)](#)

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 ORR, rotate right with extend”, “A1 ORR, shift or rotate by value”, “A1 ORRS, rotate right with extend”, and “A1 ORRS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the ORRS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T2 ORR, rotate right with extend”, “T2 ORR, shift or rotate by value”, “T2 ORRS, rotate right with extend”, and “T2 ORRS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 ORR, rotate right with extend”, “A1 ORR, shift or rotate by value”, “A1 ORRS, rotate right with extend”, and “A1 ORRS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 ORR, rotate right with extend”, “T2 ORR, shift or rotate by value”, “T2 ORRS, rotate right with extend”, and “T2 ORRS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 ORR, rotate right with extend”, “A1 ORR, shift or rotate by value”, “A1 ORRS, rotate right with extend”, and “A1 ORRS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 ORR, rotate right with extend”, “T2 ORR, shift or rotate by value”, “T2 ORRS, rotate right with extend”, and “T2 ORRS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 ORR, shift or rotate by value” and “A1 ORRS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm5” field as <amount> modulo 32.

For the “T2 ORR, shift or rotate by value” and “T2 ORRS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

In T32 assembly:

- Outside an IT block, if ORRS <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORRS <Rd>, <Rn> had been written.
- Inside an IT block, if ORR<c> <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORR<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .w qualifier.

Operation

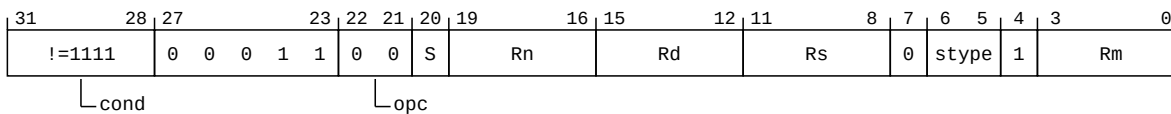
```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C[32](R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR shifted;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.N = result[31];
            PSTATE.Z = IsZeroBit[32](result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
        end;
    end;
end;
end;

```

C5.1.130 ORR, ORRS (register-shifted register)

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

```
ORR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Encoding for the Not flag setting variant

Applies when (S == 0)

```
ORR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

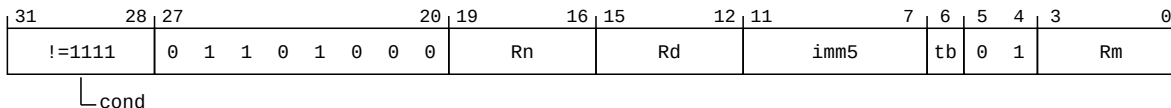
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) OR shifted;
    R(d) = result;
    if setflags then
        PSTATE.N = result[31];
        PSTATE.Z = IsZeroBit{32}(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
    end;
end;
```

C5.1.131 PKHBT, PKHTB

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the PKHBT variant

Applies when (tb == 0)

```
PKHBT{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, LSL #<imm>}
```

Encoding for the PKHTB variant

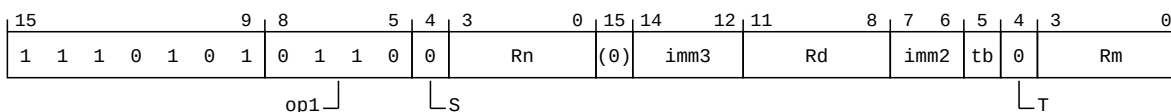
Applies when (tb == 1)

```
PKHTB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ASR #<imm>}
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let tbform : boolean = (tb == '1');
let (shift_t, shift_n) : (SRTtype, integer) = DecodeImmShift(tb::'0', imm5);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the PKHBT variant

Applies when (tb == 0)

```
PKHBT{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, LSL #<imm>} // (tbform == FALSE)
```

Encoding for the PKHTB variant

Applies when (tb == 1)

```
PKHTB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ASR #<imm>} // (tbform == TRUE)
```

Decode for all variants of this encoding

```

if S == '1' || T == '1' then Undefined(); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let tbform : boolean = (tb == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(tb::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 PKHBT” and “A1 PKHTB” variants: the shift to apply to the value read from <Rm>, encoded in the “imm5” field.

For PKHBT, it is one of:

omitted No shift, encoded as 0b000000.

1-31 Left shift by specified number of bits, encoded as a binary number.

For PKHTB, it is one of:

omitted Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn> had been written.

1-32 Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b000000. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

For the “T1 PKHBT” and “T1 PKHTB” variants: the shift to apply to the value read from <Rm>, encoded in the “imm3:imm2” field.

For PKHBT, it is one of:

omitted No shift, encoded as 0b000000.

1-31 Left shift by specified number of bits, encoded as a binary number.

For PKHTB, it is one of:

omitted Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn> had been written.

1-32 Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = Shift{ } (R(m), shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
    R(d)[15:0] = if tbform then operand2[15:0] else R(n)[15:0];
    R(d)[31:16] = if tbform then R(n)[31:16] else operand2[31:16];
end;
```

C5.1.132 PLD, PLDW (immediate)

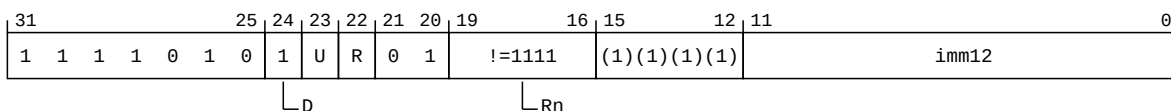
Preload Data (immediate) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Preload read variant

Applies when (R == 1)

```
PLD{<c>}{<q>}  [<Rn> {, #(<+/->)<imm>}]
```

Encoding for the Preload write variant

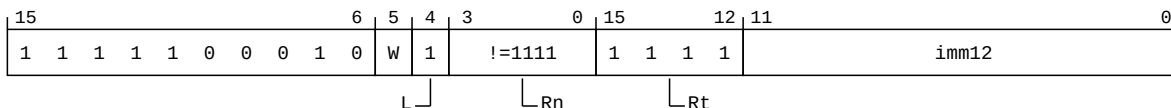
Applies when (R == 0)

```
PLDW{<c>}{<q>}  [<Rn> {, #(<+/->)<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
let is_pldw : boolean = (R == '0');
```

T1



Encoding for the Preload read variant

Applies when (W == 0)

```
PLD{<c>}{<q>}  [<Rn> {, #(<+>)<imm>}]
```

Encoding for the Preload write variant

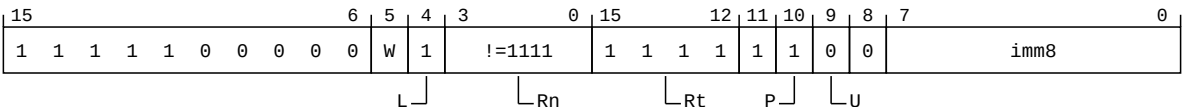
Applies when (W == 1)

```
PLDW{<c>}{<q>}  [<Rn> {, #(<+>)<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;  
let n : integer = UInt(Rn);  
let imm32 : bits(32) = ZeroExtend{}(imm12);  
let add : boolean = TRUE;  
let is_pldw : boolean = (W == '1');
```

T2



Encoding for the Preload read variant

Applies when (W == 0)

```
PLD{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Encoding for the Preload write variant

Applies when (W == 1)

```
PLDW{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Decode for all variants of this encoding

```
if Rn == '1111' then See("PLD (literal)"); end;  
let n : integer = UInt(Rn);  
let imm32 : bits(32) = ZeroExtend{}(imm8);  
let add : boolean = FALSE;  
let is_pldw : boolean = (W == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1 Preload read” and “A1 Preload write” variants: see [Standard assembler syntax fields](#). Must be AL or omitted.

For the “T1 Preload read”, “T1 Preload write”, “T2 Preload read”, and “T2 Preload write” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If the PC is used, see [PLD \(literal\)](#).

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Preload read” and “A1 Preload write” variants: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T1 Preload read” and “T1 Preload write” variants: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T2 Preload read” and “T2 Preload write” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
    end;
end;
```

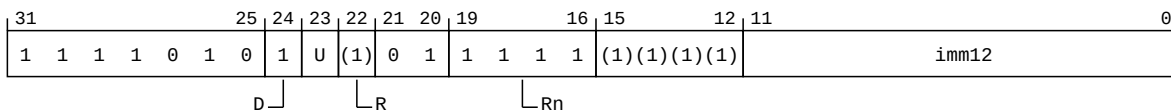
C5.1.133 PLD (literal)

Preload Data (literal) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The effect of a PLD instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

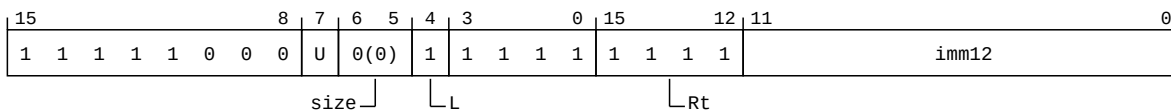
```
PLD{<c>}{<q>} <label> // (Normal form)
```

```
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

T1



Encoding

```
PLD{<c>}{<q>} <label> // (Preferred syntax)
```

```
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). Must be AL or omitted.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<label>

The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. The offset must be in the range -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

For the “T1” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = (if add then (AlignDownSize(PC32(),4) + imm32)
                             else (AlignDownSize(PC32(),4) - imm32));
    Hint_PreloadData(address);
end;
```

C5.1.134 PLD, PLDW (register)

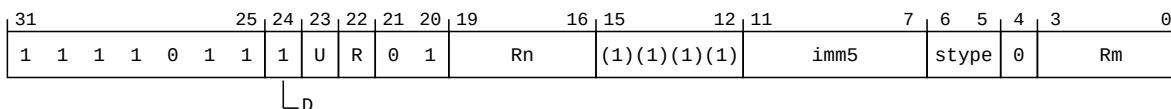
Preload Data (register) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.

The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Preload read, optional shift or rotate variant

Applies when (R == 1 && !(imm5 == 00000 && stype == 11))

```
PLD{<c>}{<q>}  [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Encoding for the Preload read, rotate right with extend variant

Applies when (R == 1 && imm5 == 00000 && stype == 11)

```
PLD{<c>}{<q>}  [<Rn>, {+/-}<Rm> , RRX]
```

Encoding for the Preload write, optional shift or rotate variant

Applies when (R == 0 && !(imm5 == 00000 && stype == 11))

```
PLDW{<c>}{<q>}  [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Encoding for the Preload write, rotate right with extend variant

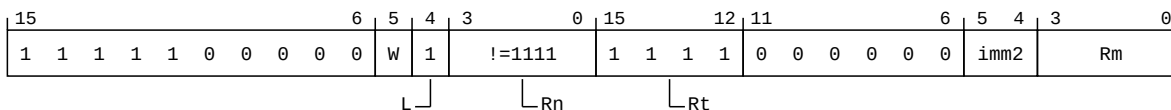
Applies when (R == 0 && imm5 == 00000 && stype == 11)

```
PLDW{<c>}{<q>}  [<Rn>, {+/-}<Rm> , RRX]
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = (U == '1');
let is_pldw : boolean = (R == '0');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
if m == 15 || (n == 15 && is_pldw) then UnpredictableProcedure(); end;
```

T1



Encoding for the Preload read variant

Applies when $(W == 0)$

```
PLD{<c>}{<q>}[<Rn>,{+<Rm>{, LSL #<amount>}]
```

Encoding for the Preload write variant

Applies when ($\bar{W} == 1$)

```
PLDW{<c>}{<q>}    [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Decode for all variants of this encoding

```

if Rn == '1111' then See("PLD (literal)"); end;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = TRUE;
let is_pldw : boolean = (W == '1');
let shift_t : SRTtype = SRTtype_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “Preload read, optional shift or rotate”, “Preload read, rotate right with extend”, “Preload write, optional shift or rotate”, and “Preload write, rotate right with extend” variants: see [Standard assembler syntax fields](#). <c> must be AL or omitted.

For the “Preload read” and “Preload write” variants: see [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rn>

For the “Preload read, optional shift or rotate”, “Preload read, rotate right with extend”, “Preload write, optional shift or rotate”, and “Preload write, rotate right with extend” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used.

For the “Preload read” and “Preload write” variants: is the general-purpose base register, encoded in the “Rn” field.

 $+/ -$

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”.

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the index register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “Preload read, optional shift or rotate” and “Preload write, optional shift or rotate” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “Preload read” and “Preload write” variants: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the “imm2” field.

+

Specifies the index register is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let address : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
    end;
end;
```

C5.1.135 PLI (immediate, literal)

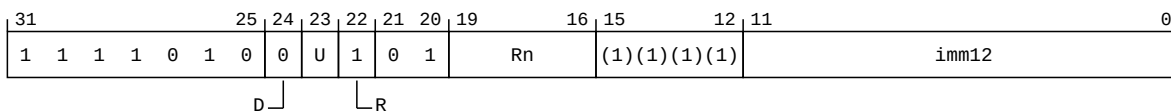
Preload Instruction (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding

PLI{<c>}{<q>} [<Rn> {, #(<+/->)<imm>}] // (Preferred syntax)

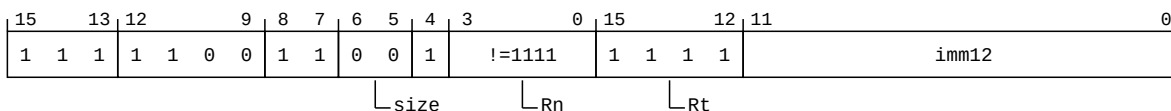
PLI{<c>}{<q>} <label> // (Normal form)

PLI{<c>}{<q>} [PC, #(<+/->)<imm>] // (Alternative form)

Decode for this encoding

```
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

T1



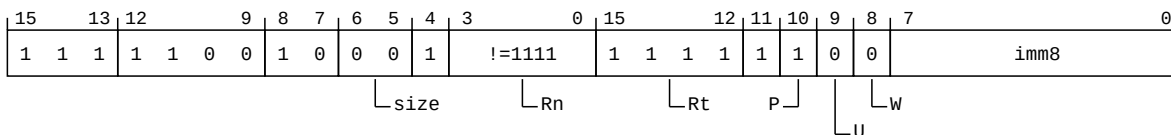
Encoding

PLI{<c>}{<q>} [<Rn> {, #(<+>)<imm>}]

Decode for this encoding

```
if Rn == '1111' then See("encoding T3"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = TRUE;
```

T2



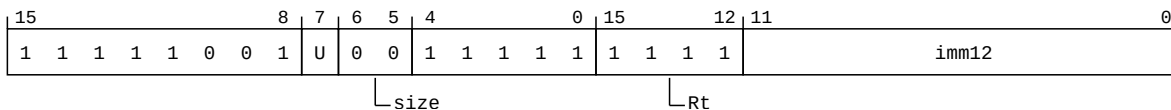
Encoding

```
PLI{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then See("encoding T3"); end;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let add : boolean = FALSE;
```

T3



Encoding

```
PLI{<c>}{<q>} <label> // (Preferred syntax)
```

```
PLI{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
let n : integer = 15;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let add : boolean = (U == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). Must be AL or omitted.

For the “T1”, “T2”, and “T3” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T1” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T2” variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

For the “T3” variant: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the “imm12” field.

<label>

The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. The offset must be in the range -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

+

Specifies the offset is added to the base register.

For the literal forms of the instruction, encoding T3 is used, or `Rn` is encoded as `0b1111` in encoding A1, to indicate that the PC is the base register.

The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
end;
```

C5.1.136 PLI (register)

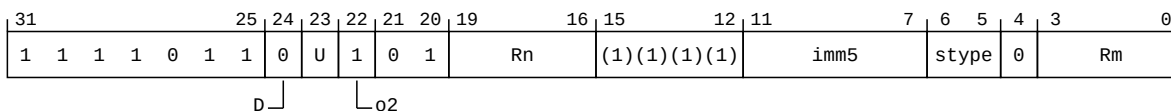
Preload Instruction (register)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see [Preloading caches](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

```
PLI{<c>}{<q>}  [<Rn>, {+/-}<Rm> , RRX]
```

Encoding for the Shift or rotate by value variant

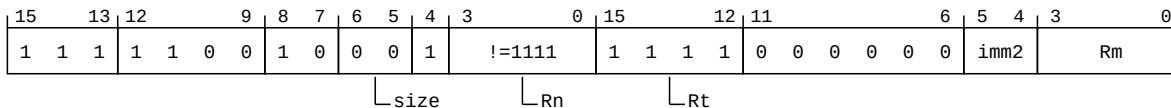
Applies when (!(imm5 == 00000 && stype == 11))

```
PLI{<c>}{<q>}  [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let add : boolean = (U == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
PLI{<c>}{<q>}  [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Decode for this encoding

```
if Rn == '1111' then See("PLI (immediate, literal)"); end;  
let n : integer = UInt(Rn);  
let m : integer = UInt(Rm);  
let add : boolean = TRUE;  
let shift_t : SRTYPE = SRTYPE_LSL;  
let shift_n : integer = UInt(imm2);  
// Armv8-A removes UNPREDICTABLE for R13  
if m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “Rotate right with extend” and “Shift or rotate by value” variants: see [Standard assembler syntax fields](#).
<c> must be AL or omitted.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the index register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T1” variant: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the “imm2” field.

+

Specifies the index register is added to the base register.

Operation

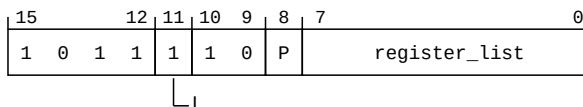
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let address : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    Hint_PreloadInstr(address);
end;
```

C5.1.137 POP

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).



Encoding

```
POP{<c>}{<q>} <registers> // (Preferred syntax)
```

```
LDM{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

Decode for this encoding

```
let registers : bits(16) = P::'0000000'::register_list;
let UnalignedAllowed : boolean = FALSE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
if registers[15] == '1' && InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<registers>

Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the “register_list” field, and can optionally include the PC. If the PC is in the list, the “P” field is set to 1, otherwise this field defaults to 0.

If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
var address : bits(32) = R(13);
for i = 0 to 14 do
    if registers[i] == '1' then
        R(i) = if UnalignedAllowed then MemU{32}(address) else MemA{32}(address);
        address = address + 4;
    end;
end;
if registers[15] == '1' then
    if UnalignedAllowed then
        if address[1:0] == '00' then
            LoadWritePC(MemU{32}(address));
        else
            UnpredictableProcedure();
        end;
    else
        LoadWritePC(MemA{32}(address));
    end;
end;
if registers[13] == '0' then R(13) = R(13) + 4*BitCount(registers); end;
if registers[13] == '1' then R(13) = ARBITRARY : bits(32); end;
end;
```

C5.1.138 POP (multiple registers)

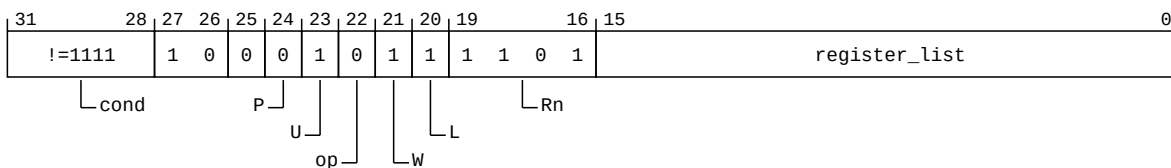
Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

This is an alias of [LDM](#), [LDMIA](#), [LDMFD](#). This means:

- The encodings in this description are named to match the encodings of [LDM](#), [LDMIA](#), [LDMFD](#).
- The description of [LDM](#), [LDMIA](#), [LDMFD](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T2](#)).

A1



Encoding for the A1 variant

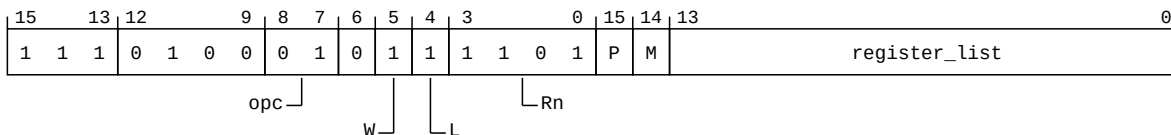
```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{IA}{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

T2



Encoding for the T2 variant

```
POP{<c>}{<q>} <registers>
```

```
POP{<c>}.W <registers> // (M == 0 && register_list[13:8] == '000000')
```

is equivalent to

```
LDM{IA}{<c>}.W SP!, <registers>
```

and is the preferred disassembly when `BitCount(P :: M :: register_list) > 1`.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<registers>

For the “A1” variant: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

If the SP is in the list, the value of the SP after such an instruction is UNKNOWN.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

Arm deprecates the use of this instruction with both the LR and the PC in the list.

For the “T2” variant: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain one of the LR or the PC. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0. If the PC is in the list, the “P” field is set to 1, otherwise it defaults to 0.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

The description of [LDM](#), [LDMIA](#), [LDMFD](#) gives the operational pseudocode for this instruction.

C5.1.139 POP (single register)

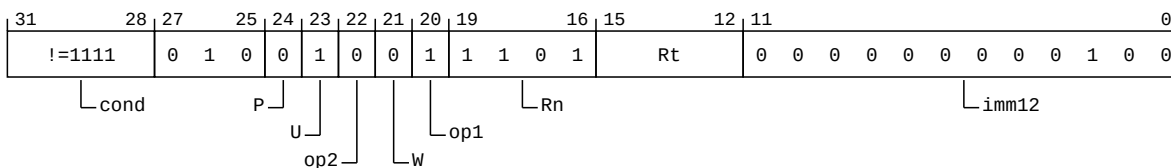
Pop Single Register from Stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point just above the loaded data.

This is an alias of [LDR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [LDR \(immediate\)](#).
- The description of [LDR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T4](#)).

A1



Encoding for the Post-indexed variant

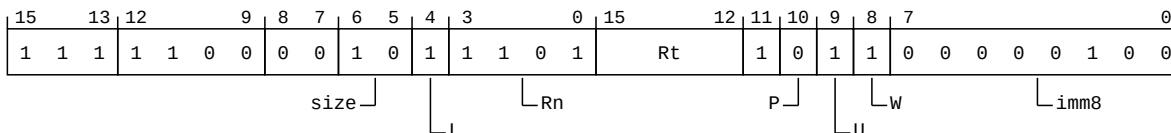
POP{<c>}{<q>} <single_register_list>

is equivalent to

LDR{<c>}{<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

T4



Encoding for the Post-indexed variant

POP{<c>}{<q>} <single_register_list>

is equivalent to

LDR{<c>}{<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<single_register_list>

Is the general-purpose register <Rt> to be loaded surrounded by { and }.

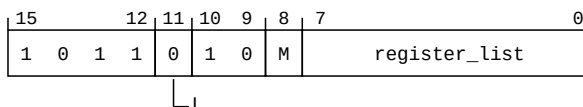
Operation

The description of [LDR \(immediate\)](#) gives the operational pseudocode for this instruction.

C5.1.140 PUSH

Push Multiple Registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).



Encoding

```
PUSH{<c>}{<q>} <registers> // (Preferred syntax)
```

```
STMDB{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

Decode for this encoding

```
let registers : bits(16) = '0'::M::'000000'::register_list;
let UnalignedAllowed : boolean = FALSE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<registers>

Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R7, encoded in the “register_list” field, and can optionally include the LR. If the LR is in the list, the “M” field is set to 1, otherwise this field defaults to 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(13) - 4*BitCount(registers);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == 13 && i != LowestSetBit(registers) then // Only possible for encoding A1
                MemA{32}(address) = ARBITRARY : bits(32);
            else
                if UnalignedAllowed then
                    MemU{32}(address) = R(i);
                else
                    MemA{32}(address) = R(i);
                end;
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then // Only possible for encoding A1 or A2
        if UnalignedAllowed then
            MemU{32}(address) = PCStoreValue();
        else
            MemA{32}(address) = PCStoreValue();
        end;
    end;
    R(13) = R(13) - 4*BitCount(registers);
end;
```

C5.1.141 PUSH (multiple registers)

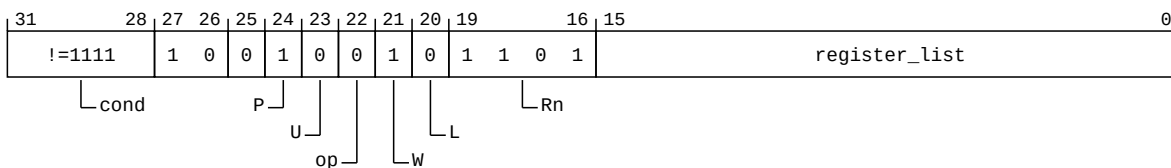
Push multiple registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

This is an alias of [STMDB](#), [STMFD](#). This means:

- The encodings in this description are named to match the encodings of [STMDB](#), [STMFD](#).
- The description of [STMDB](#), [STMFD](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the A1 variant

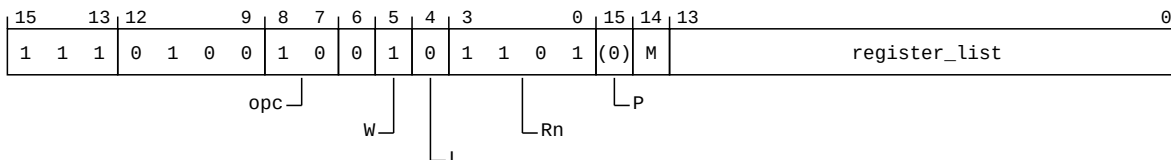
PUSH{<c>}{<q>} <registers>

is equivalent to

STMDB{<c>}{<q>} SP!, <registers>

and is the preferred disassembly when `BitCount(register_list) > 1`.

T1



Encoding for the T1 variant

PUSH{<c>}{<q>} <registers>

PUSH{<c>}.W <registers> // (register_list[13:8] == '000000')

is equivalent to

STMDB{<c>}{<q>} SP!, <registers>

and is the preferred disassembly when `BitCount(M :: register_list) > 1`.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<registers>

For the “A1” variant: is a list of two or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

The SP and PC can be in the list. However:

- Arm deprecates the use of instructions that include the PC in the list.
- If the SP is in the list, and it is not the lowest-numbered register in the list, the instruction stores an UNKNOWN value for the SP.

For the “T1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain the LR. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0.

Operation

The description of [STMDB](#), [STMTD](#) gives the operational pseudocode for this instruction.

C5.1.142 PUSH (single register)

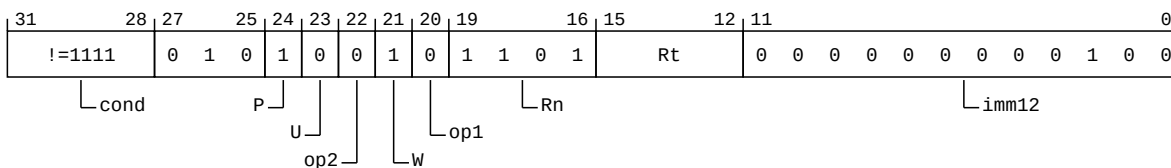
Push Single Register to Stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data.

This is an alias of [STR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [STR \(immediate\)](#).
- The description of [STR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T4](#)).

A1



Encoding for the Pre-indexed variant

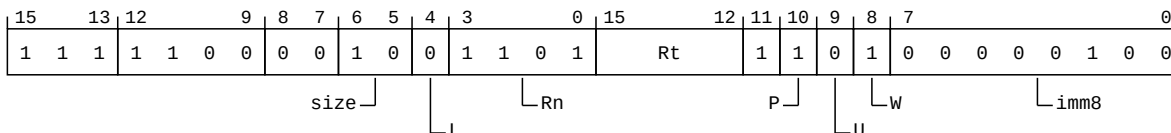
`PUSH{<c>}{<q>} <single_register_list>`

is equivalent to

`STR{<c>}{<q>} <Rt>, [SP, #-4]!`

and is always the preferred disassembly.

T4



Encoding for the Pre-indexed variant

`PUSH{<c>}{<q>} <single_register_list> // (Standard syntax)`

is equivalent to

`STR{<c>}{<q>} <Rt>, [SP, #-4]!`

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<single_register_list>

Is the general-purpose register <Rt> to be stored surrounded by { and }.

Operation

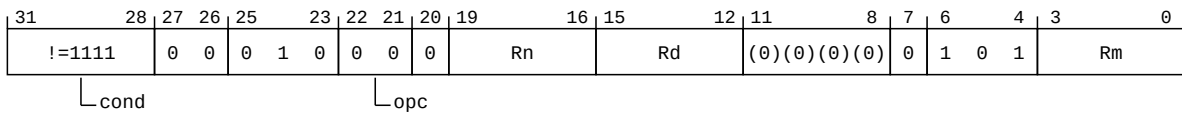
The description of [STR \(immediate\)](#) gives the operational pseudocode for this instruction.

C5.1.143 QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range -2^{31} to $(2^{31} - 1)$, and writes the result to the destination register. If saturation occurs, it sets [PSTATE.Q](#) to 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

QADD{<c>}{<q>}{<Rd>,<Rm>,<Rn>}

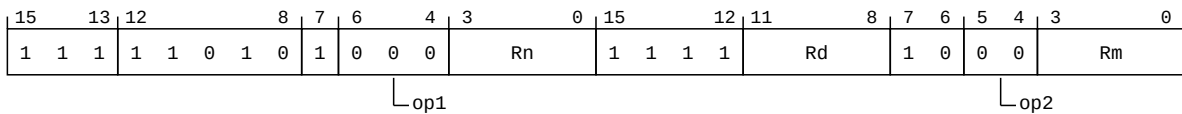
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

QADD{<c>}{<q>}{<Rd>,<Rm>,<Rn>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rn>

Is the second general-purpose source register, encoded in the “Rn” field.

Operation

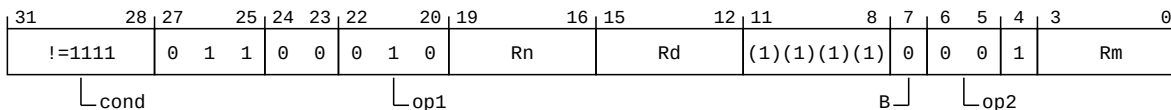
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (r, sat) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) + SInt(R(n)));
    R(d) = r;
    if sat then
        PSTATE.Q = '1';
    end;
end;
```

C5.1.144 QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



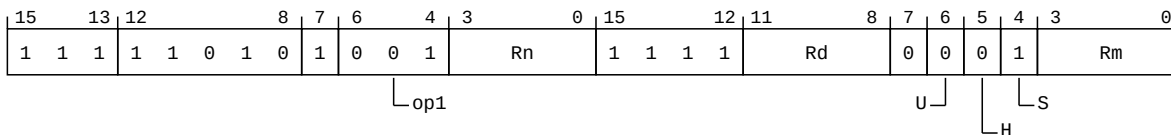
Encoding

QADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

QADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

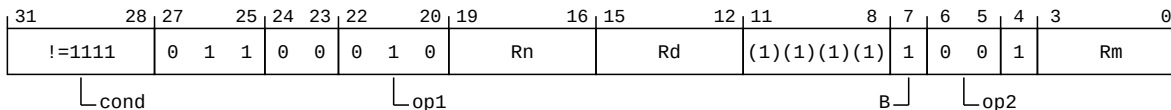
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0] = SignedSat{16}(sum1);
    R(d)[31:16] = SignedSat{16}(sum2);
end;
```

C5.1.145 QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



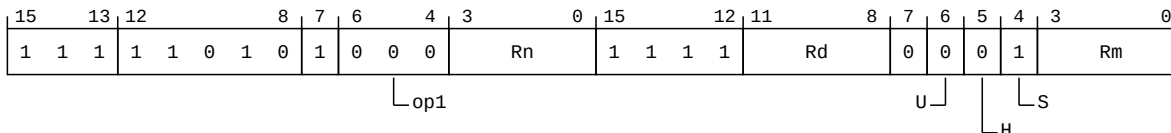
Encoding

QADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

QADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

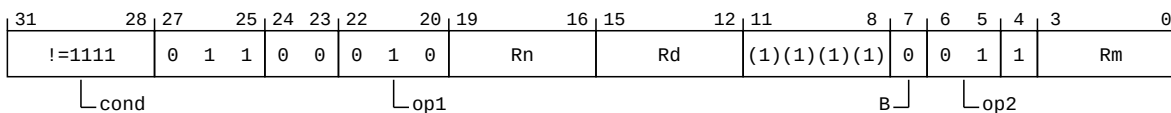
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0] = SignedSat{8}(sum1);
    R(d)[15:8] = SignedSat{8}(sum2);
    R(d)[23:16] = SignedSat{8}(sum3);
    R(d)[31:24] = SignedSat{8}(sum4);
end;
```

C5.1.146 QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



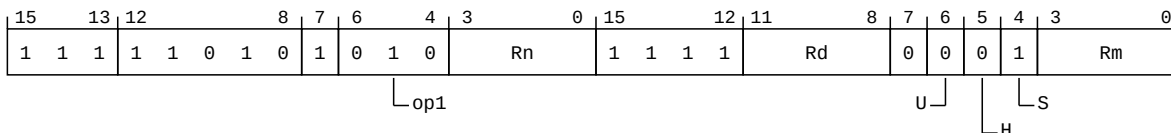
Encoding

QASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

QASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

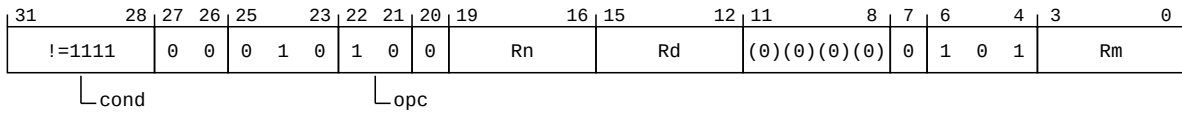
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0] = SignedSat{16}(diff);
    R(d)[31:16] = SignedSat{16}(sum);
end;
```

C5.1.147 QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation occurs in either operation, it sets [PSTATE.Q](#) to 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

QDADD{<c>}{<q>}{<Rd>,<Rm>,<Rn>}

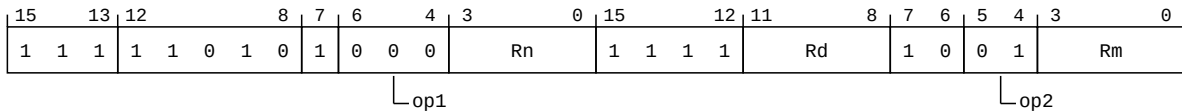
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

QDADD{<c>}{<q>}{<Rd>,<Rm>,<Rn>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rn>

Is the second general-purpose source register, encoded in the “Rn” field.

Operation

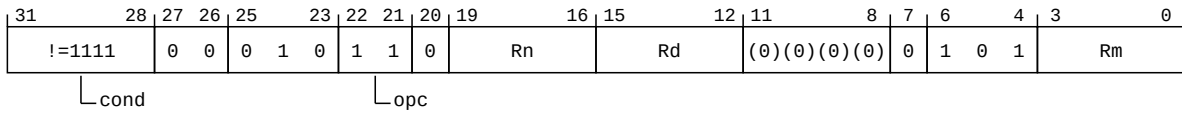
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (doubled, sat1) : (bits(32), boolean) = SignedSatQ{32}(2 * SInt(R(n)));
    let (r, sat2) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) + SInt(doubled));
    R(d) = r;
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```

C5.1.148 QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation occurs in either operation, it sets [PSTATE.Q](#) to 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

QDSUB {<c>} {<q>} {<Rd>, }<Rm>, <Rn>

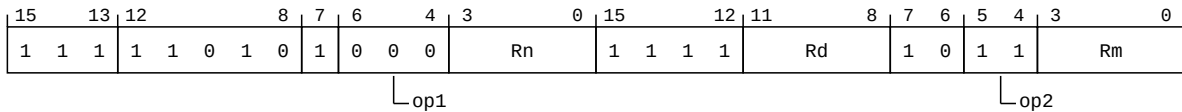
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

QDSUB{<c>}{<q>}{<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rn>

Is the second general-purpose source register, encoded in the “Rn” field.

Operation

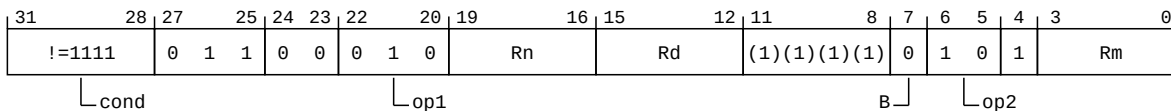
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (doubled, sat1) : (bits(32), boolean) = SignedSatQ{32}(2 * SInt(R(n)));
    let (r, sat2) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) - SInt(doubled));
    R(d) = r;
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```

C5.1.149 QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



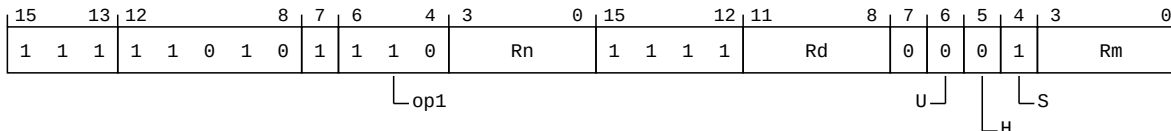
Encoding

QSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

QSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

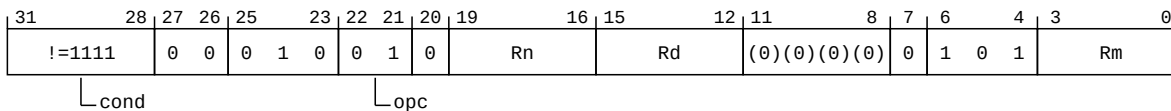
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0] = SignedSat{16}(sum);
    R(d)[31:16] = SignedSat{16}(diff);
end;
```

C5.1.150 QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets [PSTATE.Q](#) to 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



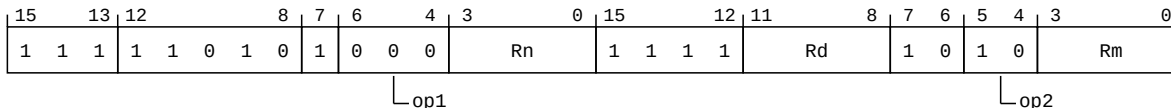
Encoding

QSUB{<c>}{<q>} {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

QSUB{<c>}{<q>} {<Rd>, }<Rm>, <Rn>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

<Rn>

Is the second general-purpose source register, encoded in the “Rn” field.

Operation

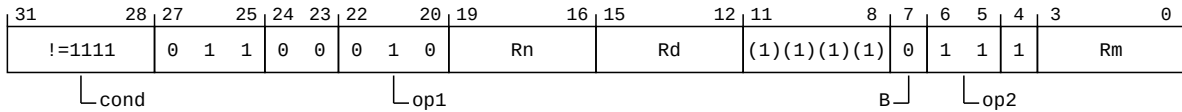
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (r, sat) : (bits(32), boolean) = SignedSatQ{32}(SInt(R(m)) - SInt(R(n)));
    R(d) = r;
    if sat then
        PSTATE.Q = '1';
    end;
end;
```

C5.1.151 QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

QSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

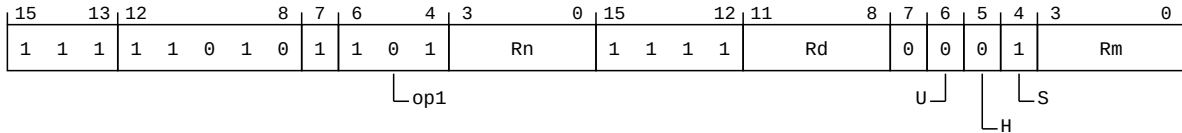
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

QSUB16{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

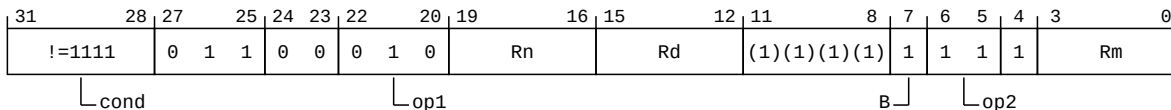
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0] = SignedSat{16}(diff1);
    R(d)[31:16] = SignedSat{16}(diff2);
end;
```

C5.1.152 QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



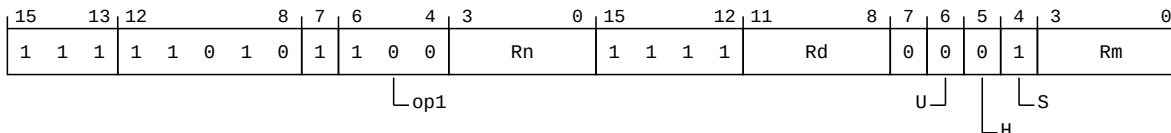
Encoding

QSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

QSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

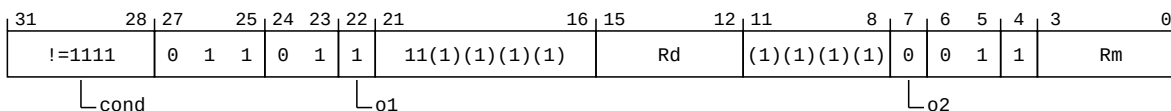
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0] = SignedSat{8}(diff1);
    R(d)[15:8] = SignedSat{8}(diff2);
    R(d)[23:16] = SignedSat{8}(diff3);
    R(d)[31:24] = SignedSat{8}(diff4);
end;
```

C5.1.153 RBIT

Reverse Bits reverses the bit order in a 32-bit register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



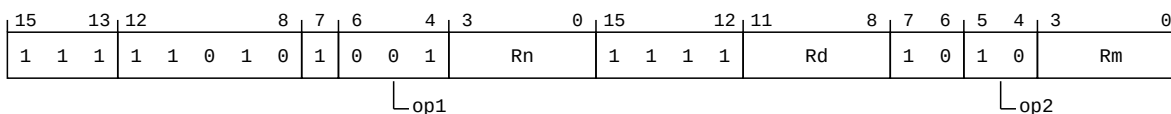
Encoding

RBIT{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

RBIT{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $m \neq n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $m = \text{UInt}(Rn)$.
- The instruction executes with the additional decode: $m = \text{UInt}(Rm)$.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” variant: is the general-purpose source register, encoded in the “Rm” field.

For the “T1” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

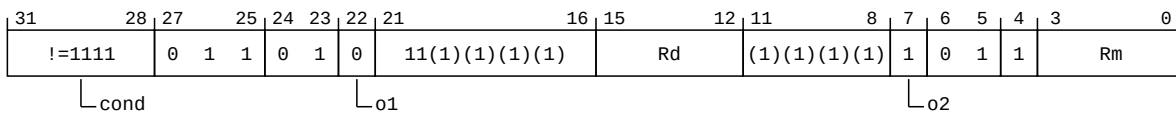
```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    for i = 0 to 31 do
        result[31-i] = R(m)[i];
    end;
    R(d) = result;
end;
```

C5.1.154 REV16

Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



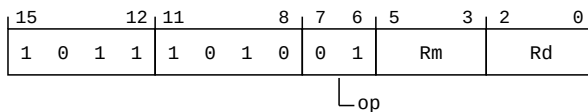
Encoding

REV16{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



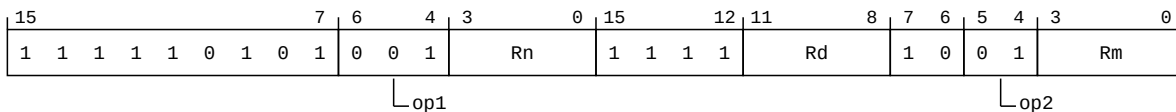
Encoding

REV16{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2



Encoding

REV16{<c>}{<q>} <Rd>, <Rm>

REV16{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $m \neq n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $m = \text{UInt}(Rn)$;
- The instruction executes with the additional decode: $m = \text{UInt}(Rm)$;
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” and “T1” variants: is the general-purpose source register, encoded in the “Rm” field.

For the “T2” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

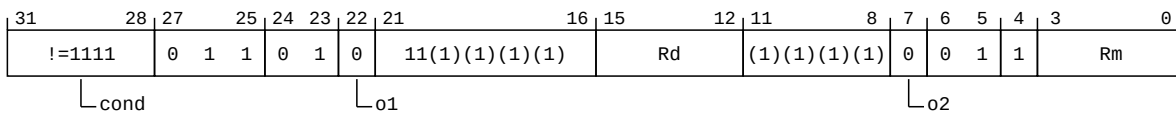
```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:24] = R(m)[23:16];
    result[23:16] = R(m)[31:24];
    result[15:8]  = R(m)[7:0];
    result[7:0]  = R(m)[15:8];
    R(d) = result;
end;
```

C5.1.155 REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



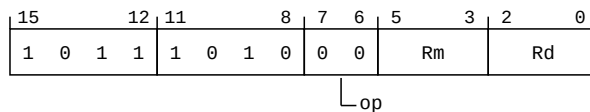
Encoding

REV{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



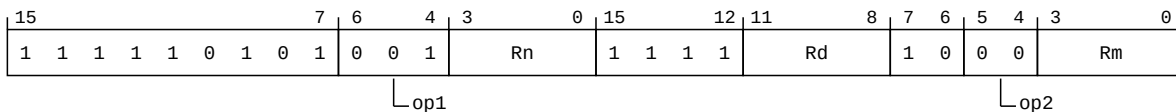
Encoding

REV{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2



Encoding

REV{<c>}{<q>} <Rd>, <Rm>

REV{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $m \neq n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $m = \text{UInt}(Rn)$;
- The instruction executes with the additional decode: $m = \text{UInt}(Rm)$;
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” and “T1” variants: is the general-purpose source register, encoded in the “Rm” field.

For the “T2” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

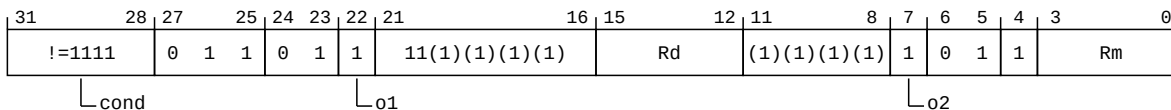
```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:24] = R(m)[7:0];
    result[23:16] = R(m)[15:8];
    result[15:8]  = R(m)[23:16];
    result[7:0]   = R(m)[31:24];
    R(d) = result;
end;
```

C5.1.156 REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



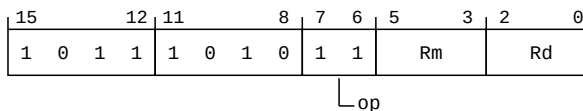
Encoding

REVSH{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



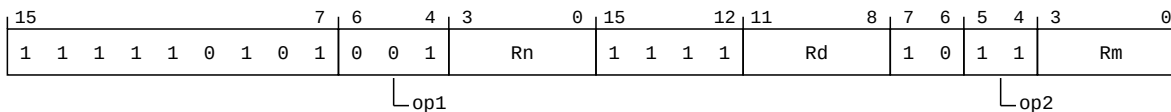
Encoding

REVSH{<c>}{<q>} <Rd>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
```

T2



Encoding

REVSH{<c>}{<q>} <Rd>, <Rm>

REVSH{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $m \neq n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $m = \text{UInt}(Rn)$;
- The instruction executes with the additional decode: $m = \text{UInt}(Rm)$;
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” and “T1” variants: is the general-purpose source register, encoded in the “Rm” field.

For the “T2” variant: is the general-purpose source register, encoded in the “Rm” field. It must be encoded with an identical value in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : bits(32);
    result[31:8] = SignExtend{24}(R(m)[7:0]);
    result[7:0]  = R(m)[15:8];
    R(d) = result;
end;
```

C5.1.157 RFE, RFEDA, RFEDB, RFEIA, RFEIB

Return From Exception loads two consecutive memory locations using an address in a base register:

- The word loaded from the lower address is treated as an instruction address. The PE branches to it.
- The word loaded from the higher address is used to restore [PSTATE](#). This word must be in the format of an SPSR.

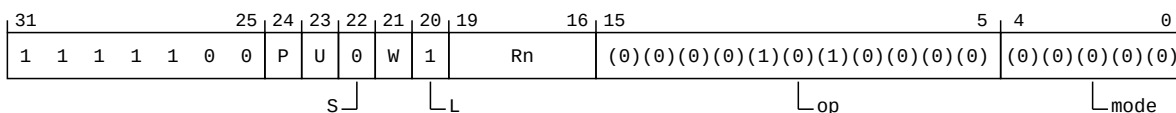
An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See [Illegal return events from AArch32 state](#).

RFE is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Decrement After variant

Applies when (P == 0 && U == 0)

RFEDA{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEEFA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Full Ascending stack)

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0)

RFEDB{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEEBA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Empty Ascending stack)

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

RFEIA{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEIFA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Full Descending stack)

Encoding for the Increment Before variant

Applies when (P == 1 && U == 1)

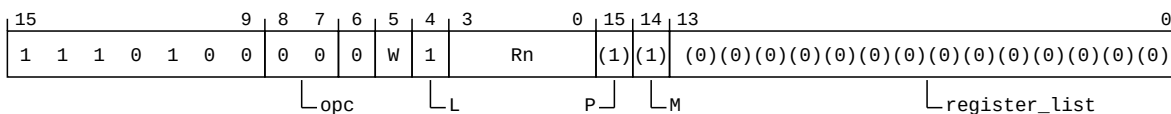
RFEIB{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEIBA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Empty Descending stack)

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

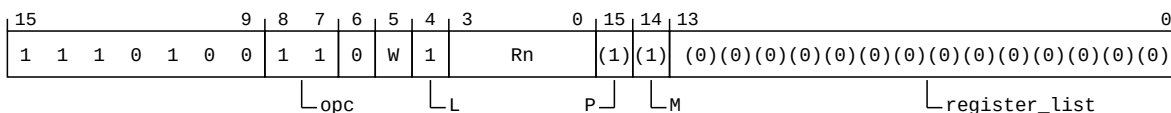
RFE^{DB}{<c>}{<q>} <Rn>{!} // (Outside or last in IT block, preferred syntax)

RFE^{FA}{<c>}{<q>} <Rn>{!} // (Outside or last in IT block, alternate syntax, Full Ascending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = FALSE;
let wordhigher : boolean = FALSE;
if n == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

T2



Encoding

RFE{IA}{<c>}{<q>} <Rn>{!} // (Outside or last in IT block, preferred syntax)

RFE^{FD}{<c>}{<q>} <Rn>{!} // (Outside or last in IT block, alternate syntax, Full Descending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let wback : boolean = (W == '1');
let increment : boolean = TRUE;
let wordhigher : boolean = FALSE;
if n == 15 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “Decrement After”, “Decrement Before”, “Increment After”, and “Increment Before” variants: see [Standard assembler syntax fields](#). <c> must be AL or omitted.

For the “T1” and “T2” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

IA

For the “Increment After” variant: is an optional suffix to indicate the Increment After variant.

For the “T2” variant: is an optional suffix for the Increment After form.

RFEFA, RFEEA, RFEFD, and RFEED are pseudo-instructions for RFEDA, RFEDB, RFEIA, and RFEIB respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elseif PSTATE.EL == EL0 then
        UnpredictableProcedure();           // UNDEFINED or NOP
    else
        var address : bits(32) = if increment then R(n) else R(n)-8;
        if wordhigher then address = address+4; end;
        let new_pc_value : bits(32) = MemA{32}(address);
        let spsr : bits(32) = MemA{32}(address+4);
        if wback then R(n) = if increment then R(n)+8 else R(n)-8; end;
        AArch32_ExceptionReturn(new_pc_value, spsr);
    end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

C5.1.158 ROR (immediate)

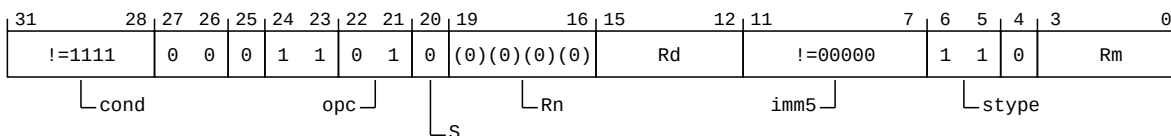
Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T3](#)).

A1



Encoding for the MOV, shift or rotate by value variant

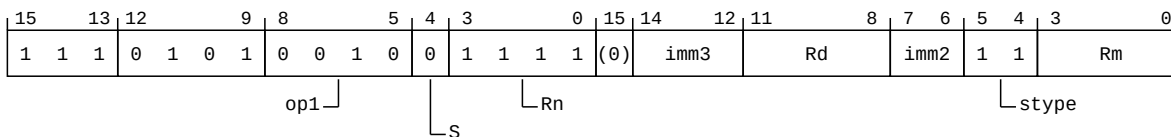
ROR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

T3



Encoding for the MOV, shift or rotate by value variant

Applies when (! (imm3 == 000 && imm2 == 00))

ROR{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T3 MOV, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 MOV, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOV, shift or rotate by value” variant: is the shift amount, in the range 1 to 31, encoded in the “imm5” field.

For the “T3 MOV, shift or rotate by value” variant: is the shift amount, in the range 1 to 31, encoded in the “imm3:imm2” field.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.159 ROR (register)

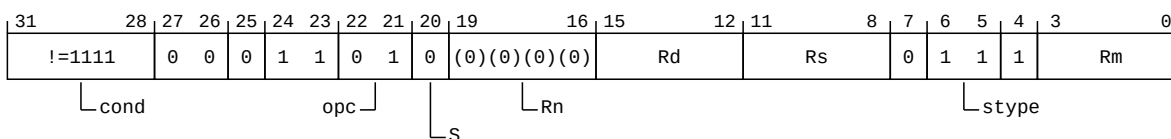
Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Not flag setting variant

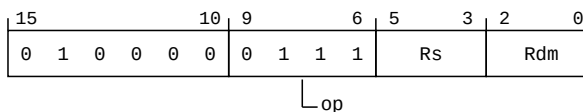
ROR{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

T1



Encoding for the Rotate right variant

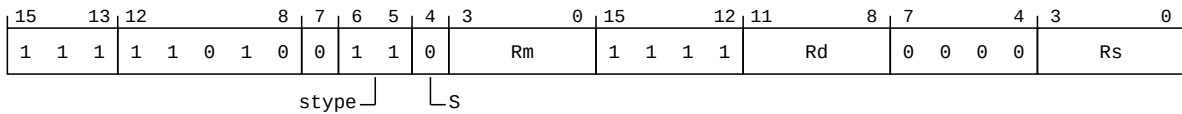
ROR<c>{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>

and is the preferred disassembly when `InITBlock()`.

T2



Encoding for the Not flag setting variant

ROR{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

```
ROR<c>.W {<Rd>, }<Rm>, <Rs> // (InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

 $\langle R_s \rangle$

Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

C5.1.160 RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

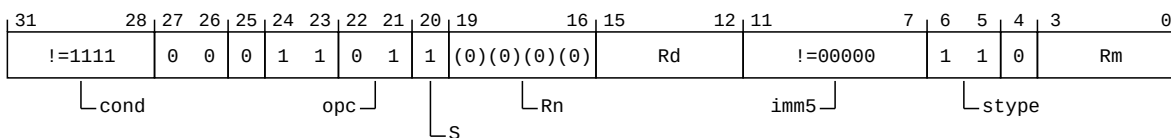
- The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVs \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVs \(register\)](#).
- The description of [MOV, MOVs \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T3](#)).

A1



Encoding for the MOVs, shift or rotate by value variant

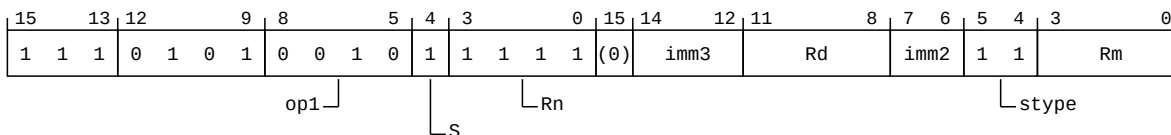
RORS{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

T3



Encoding for the MOVs, shift or rotate by value variant

Applies when (! (imm3 == 000 && imm2 == 00))

RORS{<c>}{<q>} {<Rd>, }<Rm>, #<imm>

is equivalent to

MOVS{<c>} {<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the “T3 MOVS, shift or rotate by value” variant: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 MOVS, shift or rotate by value” variant: is the general-purpose source register, encoded in the “Rm” field.

<imm>

For the “A1 MOVS, shift or rotate by value” variant: is the shift amount, in the range 1 to 31, encoded in the “imm5” field.

For the “T3 MOVS, shift or rotate by value” variant: is the shift amount, in the range 1 to 31, encoded in the “imm3:imm2” field.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.161 RORS (register)

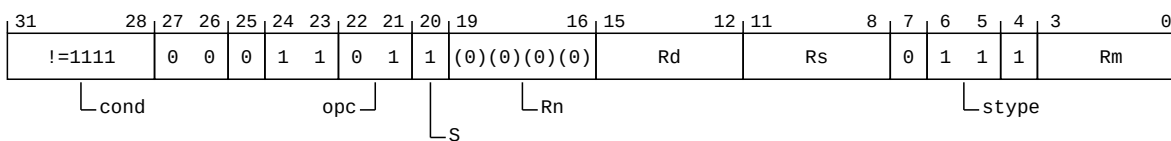
Rotate Right, setting flags (register) provides the value of the contents of a register rotated by a variable number of bits, and updates the condition flags based on the result. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register.

This is an alias of [MOV, MOVS \(register-shifted register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Flag setting variant

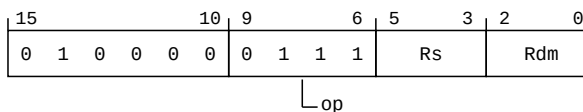
RORS{<c>}{<q>} {<Rd>, }<Rm>, <Rs>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

T1



Encoding for the Rotate right variant

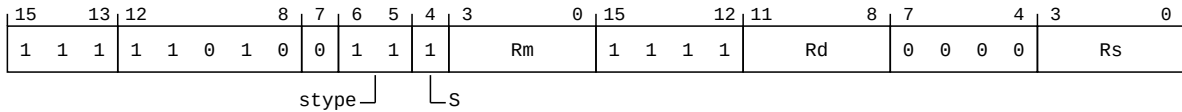
RORS{<q>} {<Rdm>, }<Rdm>, <Rs>

is equivalent to

MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>

and is the preferred disassembly when !InITBlock().

T2



Encoding for the Flag setting variant

RORS { <c> } { <q> } { <Rd>, } <Rm>, <Rs>

```
RORS.W {<Rd>, }<Rm>, <Rs> // (!InITBlock() && (UInt(Rd) < 8 && Rd == Rm && UInt(Rs) < 8))
```

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>

and is always the preferred disassembly.

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the first general-purpose source register, encoded in the “Rm” field.

 $\langle R_s \rangle$

Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the “Rs” field.

<Rdm>

Is the first general-purpose source register and the destination register, encoded in the “Rdm” field.

Operation

The description of **MOV, MOVS (register-shifted register)** gives the operational pseudocode for this instruction.

C5.1.162 RRX

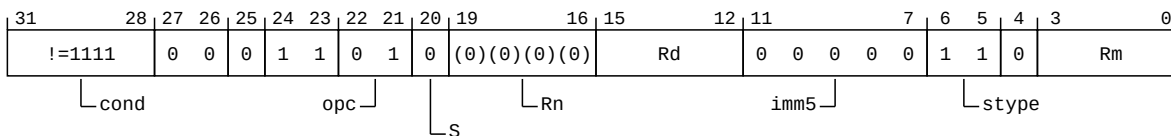
Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T3](#)).

A1



Encoding for the MOV, rotate right with extend variant

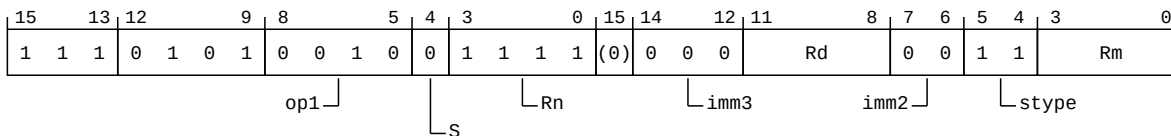
RRX{<c>}{<q>}{<Rd>,<Rm>}

is equivalent to

MOV{<c>}{<q>}{<Rd>,<Rm>},RRX

and is always the preferred disassembly.

T3



Encoding for the MOV, rotate right with extend variant

RRX{<c>}{<q>}{<Rd>,<Rm>}

is equivalent to

MOV{<c>}{<q>}{<Rd>,<Rm>},RRX

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOV, rotate right with extend” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T3 MOV, rotate right with extend” variant: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOV, rotate right with extend” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 MOV, rotate right with extend” variant: is the general-purpose source register, encoded in the “Rm” field.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.163 RRXS

Rotate Right with Extend, setting flags provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

If the destination register is not the PC, this instruction updates the condition flags based on the result, and bit[0] is shifted into the Carry flag.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

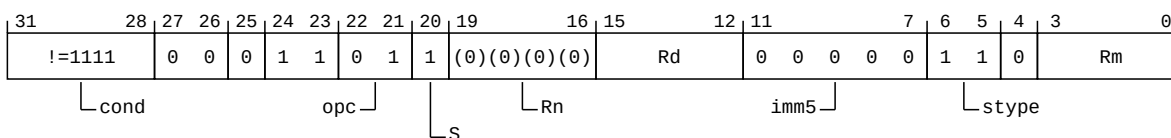
- The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of [MOV, MOVS \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T3](#)).

A1



Encoding for the MOVS, rotate right with extend variant

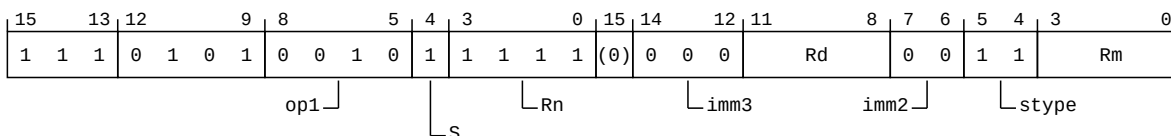
RRXS{<c>}{<q>} {<Rd>, }<Rm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, RRX

and is always the preferred disassembly.

T3



Encoding for the MOVS, rotate right with extend variant

RRXS{<c>}{<q>} {<Rd>, }<Rm>

is equivalent to

MOVS{<c>}{<q>} <Rd>, <Rm>, RRX

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 MOVS, rotate right with extend” variant: is the general-purpose destination register, encoded in the “Rd” field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores [PSTATE](#) from SPSR_<current_mode>.

For the “T3 MOVS, rotate right with extend” variant: is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1 MOVS, rotate right with extend” variant: is the general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T3 MOVS, rotate right with extend” variant: is the general-purpose source register, encoded in the “Rm” field.

Operation

The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode for this instruction.

C5.1.164 RSB, RSBS (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register.

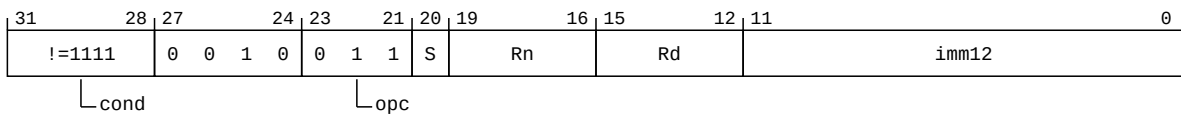
If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the RSB variant

Applies when (S == 0)

```
RSB{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Encoding for the RSBS variant

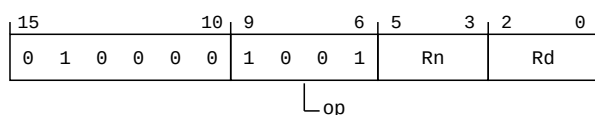
Applies when (S == 1)

```
RSBS{<c>}{<q>} {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1



Encoding

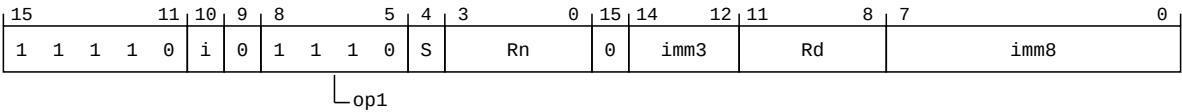
RSB<c>{<q>} {<Rd>, }<Rn>, #0 // (InITBlock())

RSBS{<q>} {<Rd>, }<Rn>, #0 // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = Zeros{}; // immediate = #0
```

T2



Encoding for the RSB variant

Applies when (S == 0)

RSB{<c>}{<q>} {<Rd>, }<Rn>, #<const>

RSB<c>.W {<Rd>, }<Rn>, #0 // (Inside IT block)

Encoding for the RSBS variant

Applies when (S == 1)

RSBS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

RSBS.W {<Rd>, }<Rn>, #0 // (Outside IT block)

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 RSB” and “A1 RSBS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the RSBS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T1”, “T2 RSB”, and “T2 RSBS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 RSB” and “A1 RSBS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 RSB”, and “T2 RSBS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 RSB” and “A1 RSBS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T2 RSB” and “T2 RSBS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), imm32, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
end;

```

C5.1.165 RSB, RSBS (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register.

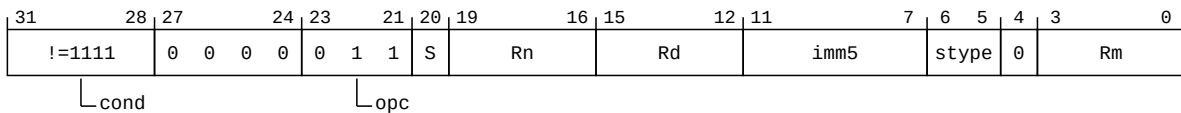
If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the RSB, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

RSB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSB, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

RSB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the RSBS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

RSBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSBS, shift or rotate by value variant

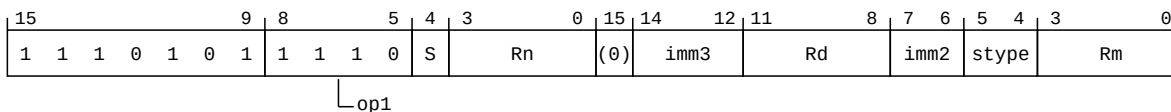
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

RSBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1



Encoding for the RSB, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

RSB { <c> } { <q> } { <Rd>, } <Rn>, <Rm>, RRX

Encoding for the RSB, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
RSB{<c>}{<q>}{<Rd>,<Rn>,<Rm>,<shift>#<amount>}
```

Encoding for the RSBS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

$$\text{RSBS}\{\langle c \rangle\}\{\langle q \rangle\}\{\langle Rd \rangle, \langle Rn \rangle, \langle Rm \rangle, \text{RRX}$$

Encoding for the RSBS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
RSBS {<c>} {<q>} {<Rd> , } <Rn> , <Rm> { , <shift> #<amount> }
```

Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 RSB, rotate right with extend”, “A1 RSB, shift or rotate by value”, “A1 RSBS, rotate right with extend”, and “A1 RSBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the RSBS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T1 RSB, rotate right with extend”, “T1 RSB, shift or rotate by value”, “T1 RSBS, rotate right with extend”, and “T1 RSBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as `<Rn>`.

<Rn>

For the “A1 RSB, rotate right with extend”, “A1 RSB, shift or rotate by value”, “A1 RSBS, rotate right with extend”, and “A1 RSBS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 RSB, rotate right with extend”, “T1 RSB, shift or rotate by value”, “T1 RSBS, rotate right with extend”, and “T1 RSBS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 RSB, rotate right with extend”, “A1 RSB, shift or rotate by value”, “A1 RSBS, rotate right with extend”, and “A1 RSBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1 RSB, rotate right with extend”, “T1 RSB, shift or rotate by value”, “T1 RSBS, rotate right with extend”, and “T1 RSBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 RSB, shift or rotate by value” and “A1 RSBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when `<shift>` = LSL or ROR) or 1 to 32 (when `<shift>` = LSR or ASR) encoded in the “imm5” field as `<amount>` modulo 32.

For the “T1 RSB, shift or rotate by value” and “T1 RSBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when `<shift>` = LSL or ROR) or 1 to 32 (when `<shift>` = LSR or ASR), encoded in the “imm3:imm2” field as `<amount>` modulo 32.

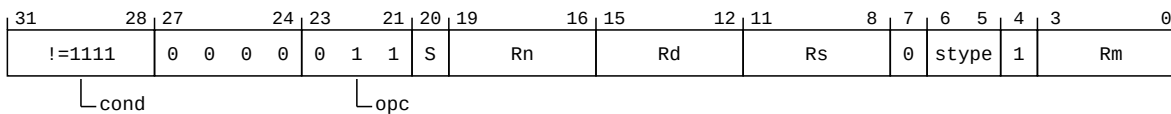
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{ }(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzc_v) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
```

```
        ALUWritePC(result);
    end;
else
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
end;
```

C5.1.166 RSB, RSBS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

RSBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

RSB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, '1');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

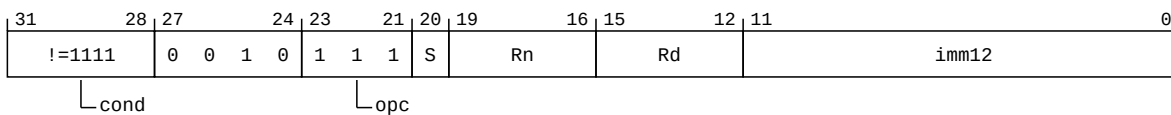
C5.1.167 RSC, RSCS (immediate)

Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.



Encoding for the RSC variant

Applies when (S == 0)

RSC{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the RSCS variant

Applies when (S == 1)

RSCS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

- For the RSCS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

<const>

An immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

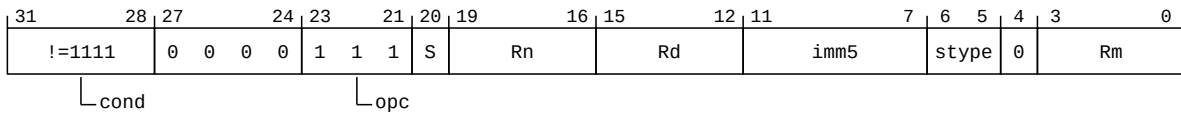
C5.1.168 RSC, RSCS (register)

Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores [PSTATE](#) from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.



Encoding for the RSC, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

RSC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSC, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

RSC{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the RSCS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

RSCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the RSCS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

RSCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the RSCS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

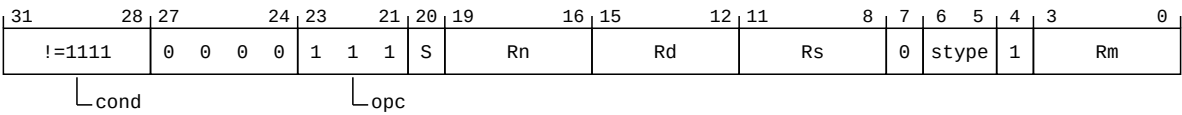
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
```

```
        PSTATE.[N,Z,C,V] = nzcvc;  
    end;  
end;  
end;
```

C5.1.169 RSC, RSCS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

RSCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

RSC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

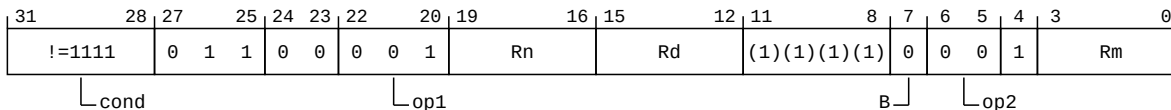
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(NOT(R(n)), shifted, PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

C5.1.170 SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results of the additions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



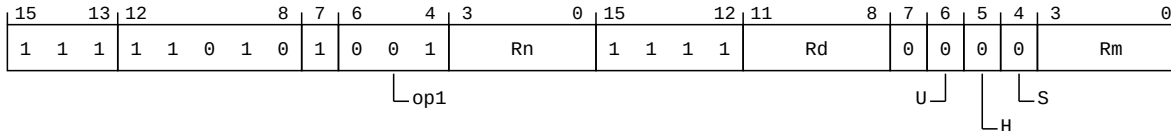
Encoding

SADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

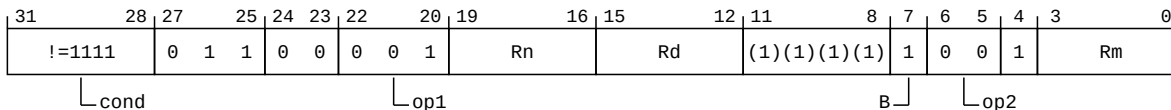
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0] = sum1[15:0];
    R(d)[31:16] = sum2[15:0];
    PSTATE.GE[1:0] = if sum1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum2 >= 0 then '11' else '00';
end;
```

C5.1.171 SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results of the additions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



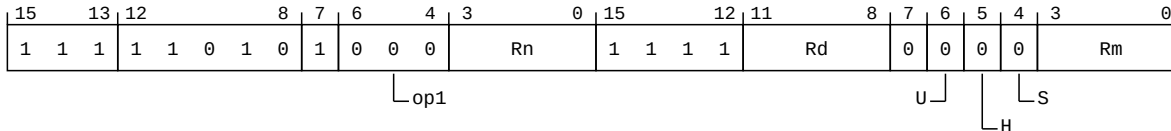
Encoding

SADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

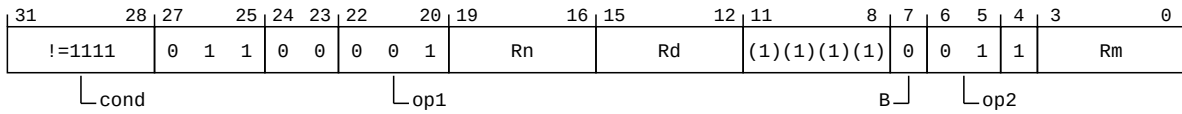
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0] = sum1[7:0];
    R(d)[15:8] = sum2[7:0];
    R(d)[23:16] = sum3[7:0];
    R(d)[31:24] = sum4[7:0];
    PSTATE.GE[0] = if sum1 >= 0 then '1' else '0';
    PSTATE.GE[1] = if sum2 >= 0 then '1' else '0';
    PSTATE.GE[2] = if sum3 >= 0 then '1' else '0';
    PSTATE.GE[3] = if sum4 >= 0 then '1' else '0';
end;
```

C5.1.172 SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets `PSTATE.GE` according to the results.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

SASX{<c>}{<q>}{<Rd>,<Rn>,<Rm>}

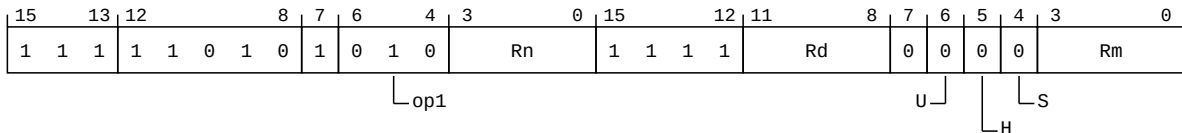
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

SASX{<c>}{<q>}{<Rd>, }<Rn>, <Rm>

Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0] = diff[15:0];
    R(d)[31:16] = sum[15:0];
    PSTATE.GE[1:0] = if diff >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum >= 0 then '11' else '00';
end;
```

C5.1.173 SBC, SBCS (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

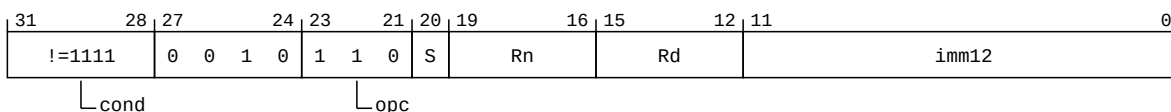
If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SBC variant

Applies when $(S == 0)$

SBC {<c>} {<q>} {<Rd>, }<Rn>, #<const>

Encoding for the SBCS variant

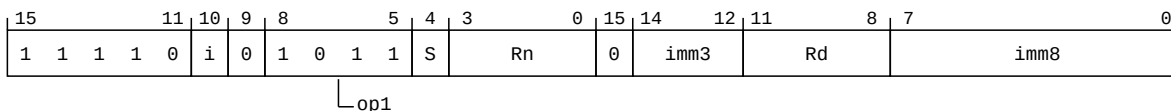
Applies when (S == 1)

```
SBCS{<c>} {<q>} {<Rd>, }<Rn>, #<const>
```

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1



Encoding for the SBC variant

Applies when $(S == 0)$

SBC{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the SBCS variant

Applies when (S == 1)

SBCS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 SBC” and “A1 SBCS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SBCS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T1 SBC” and “T1 SBCS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 SBC” and “A1 SBCS” variants: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 SBC” and “T1 SBCS” variants: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1 SBC” and “A1 SBCS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1 SBC” and “T1 SBCS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

C5.1.174 SBC, SBCS (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

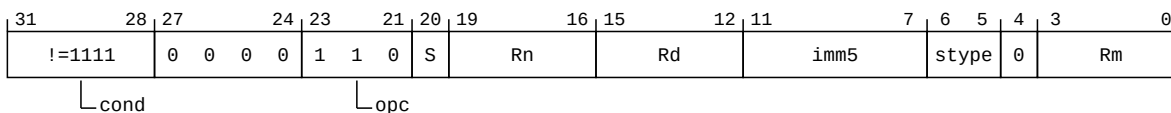
If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the SBC, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

SBC{<c>}{<q>}{<Rd>,<Rn>,<Rm>,RRX

Encoding for the SBC, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

SBC{<c>}{<q>}{<Rd>,<Rn>,<Rm>{,<shift>#<amount>}

Encoding for the SBCS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

SBCS{<c>}{<q>}{<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SBCS, shift or rotate by value variant

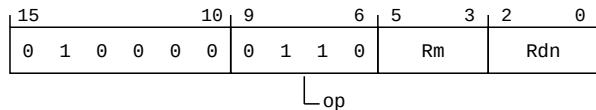
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

SBCS{<c>}{<q>}{<Rd>,<Rn>,<Rm>{,<shift>#<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift t, shift n) : (SRTyp, integer) = DecodeImmShift(stype, imm5);
```


T1



Encoding

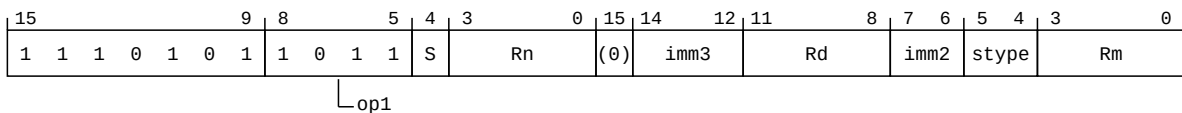
```
SBC<c>{<q>} {<Rdn>, }<Rdn>, <Rm> // (InITBlock())
```

SBCS{<q>} {<Rdn>, }<Rdn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the SBC, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

SBC {<c>} {<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SBC, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SBC{<c>}{<q>}{<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

SBC<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the SBCS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

SBCS{<c>}{<q>}{<Rd>,<Rn>,<Rm>,<RR>

Encoding for the SBCS, shift or rotate by value variant

Applies when (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SBCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

SBCS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 SBC, rotate right with extend”, “A1 SBC, shift or rotate by value”, “A1 SBCS, rotate right with extend”, and “A1 SBCS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SBCS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T2 SBC, rotate right with extend”, “T2 SBC, shift or rotate by value”, “T2 SBCS, rotate right with extend”, and “T2 SBCS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 SBC, rotate right with extend”, “A1 SBC, shift or rotate by value”, “A1 SBCS, rotate right with extend”, and “A1 SBCS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T2 SBC, rotate right with extend”, “T2 SBC, shift or rotate by value”, “T2 SBCS, rotate right with extend”, and “T2 SBCS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 SBC, rotate right with extend”, “A1 SBC, shift or rotate by value”, “A1 SBCS, rotate right with extend”, and “A1 SBCS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 SBC, rotate right with extend”, “T2 SBC, shift or rotate by value”, “T2 SBCS, rotate right with extend”, and “T2 SBCS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 SBC, shift or rotate by value” and “A1 SBCS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T2 SBC, shift or rotate by value” and “T2 SBCS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

<Rdn>

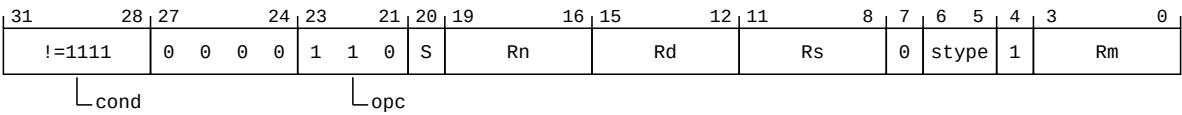
Is the first general-purpose source register and the destination register, encoded in the “Rdn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), PSTATE.C);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

C5.1.175 SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

SBCS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

SBC{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

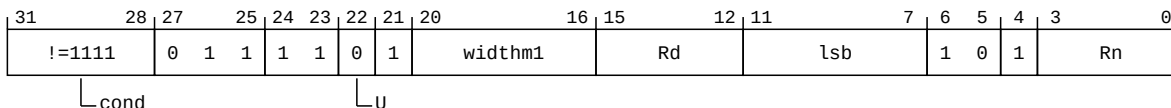
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), PSTATE.C);
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

C5.1.176 SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding

SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

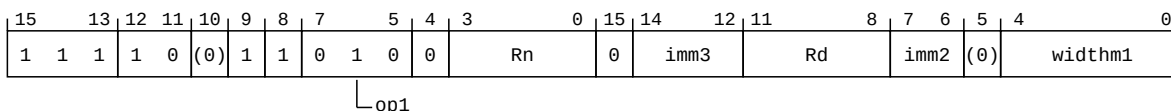
```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(lsb);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding

SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(imm3::imm2);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “A1” variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the “lsb” field.

For the “T1” variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the “imm3:imm2” field.

<width>

Is the width of the field, in the range 1 to 32-`<lsb>`, encoded in the “widthm1” field as `<width>-1`.

Operation

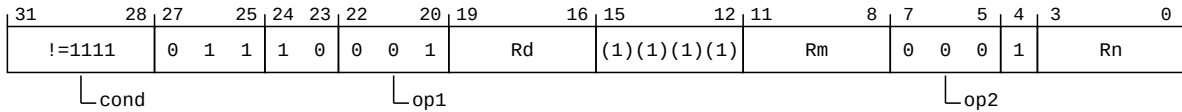
```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d) = SignExtend{32}(R(n)[msbit:lsbit]);
end;
```

C5.1.177 SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

SDIV{<c>}{<q>}{<Rd>,<Rn>,<Rm>}

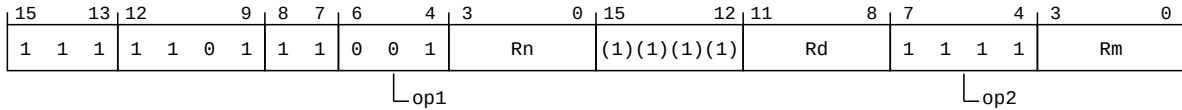
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

$$\text{SDIV}\{\langle c \rangle\}\{\langle q \rangle\} \quad \{\langle Rd \rangle, \} \langle Rn \rangle, \langle Rm \rangle$$

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the dividend, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the divisor, encoded in the “Rm” field.

Overflow

If the signed integer division $0x80000000 / 0xFFFFFFFF$ is performed, the pseudocode produces the intermediate integer result $+2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to <Rd> must be the bottom 32 bits of the binary representation of $+2^{31}$. So the result of the division is $0x80000000$.

Operation

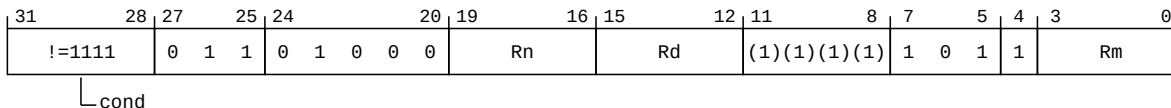
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let dividend : integer = SInt(R(n));
    let divisor : integer = SInt(R(m));
    var result : integer;
    if divisor == 0 then
        result = 0;
    elsif (dividend < 0) == (divisor < 0) then
        result = Abs(dividend) DIVRM Abs(divisor);    // same signs - positive result
    else
        result = -(Abs(dividend) DIVRM Abs(divisor)); // different signs - negative result
    end;
    R(d) = result[31:0];
end;
```

C5.1.178 SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the [PSTATE.GE](#) flags.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



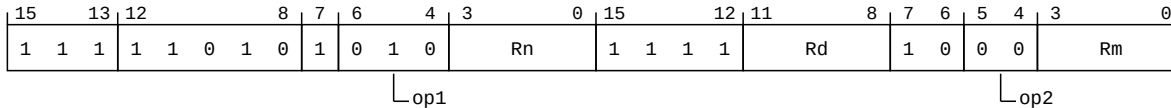
Encoding

SEL{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SEL{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

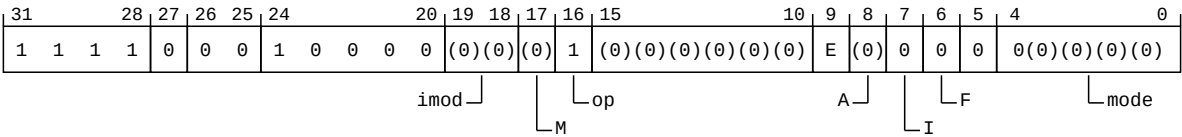
```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d)[7:0]   = if PSTATE.GE[0] == '1' then R(n)[7:0]   else R(m)[7:0];
    R(d)[15:8]  = if PSTATE.GE[1] == '1' then R(n)[15:8]  else R(m)[15:8];
    R(d)[23:16] = if PSTATE.GE[2] == '1' then R(n)[23:16] else R(m)[23:16];
    R(d)[31:24] = if PSTATE.GE[3] == '1' then R(n)[31:24] else R(m)[31:24];
end;
```

C5.1.179 SETEND

Set Endianness writes a new value to `PSTATE.E`.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



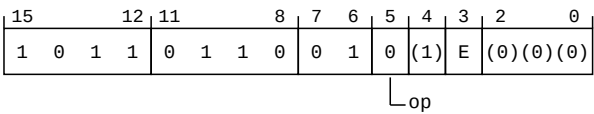
Encoding

```
SETEND(<q> <endian_specifier> // (Cannot be conditional)
```

Decode for this encoding

```
let set_bigend : boolean = (E == '1');
```

T1



Encoding

```
SETEND(<q> <endian_specifier> // (Not permitted in IT block)
```

Decode for this encoding

```
let set_bigend : boolean = (E == '1');  
if InITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<endian_specifier>

Is the endianness to be selected, and the value to be set in `PSTATE.E`, encoded in “E”:

E	<endian_specifier>
0	LE
1	BE

Operation

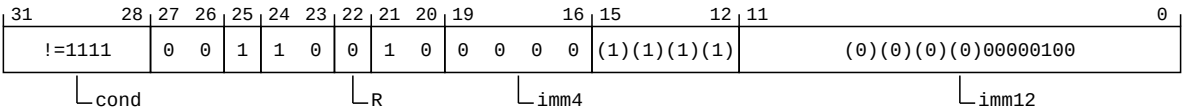
```
EncodingSpecificOperations();  
AArch32_CheckSETENDEnabled();  
PSTATE.E = if set_bigend then '1' else '0';
```

C5.1.180 SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see [Wait For Event and Send Event](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



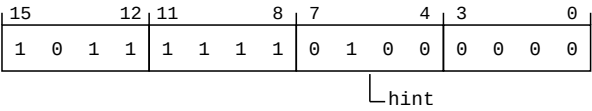
Encoding

SEV{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1



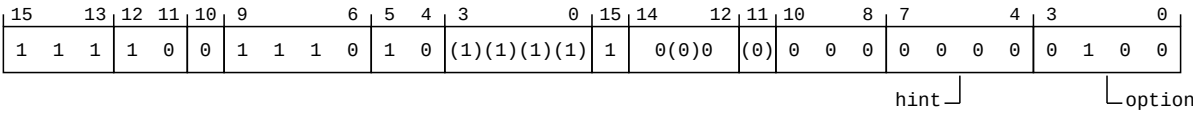
Encoding

SEV{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2



Encoding

SEV{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

Operation

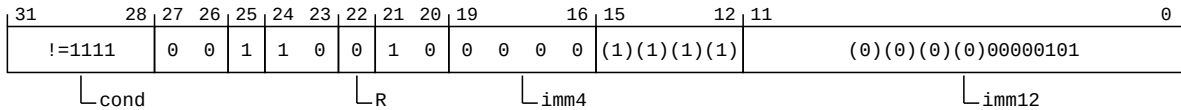
```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
end;
```

C5.1.181 SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a *WFE* instruction.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1** and **T2**).

A1



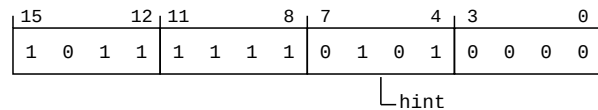
Encoding

SEVL{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1



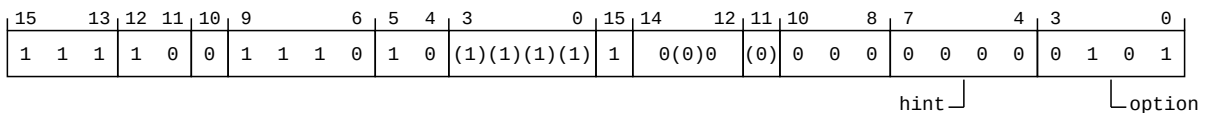
Encoding

SEVL{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2



Encoding

SEVL{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

Operation

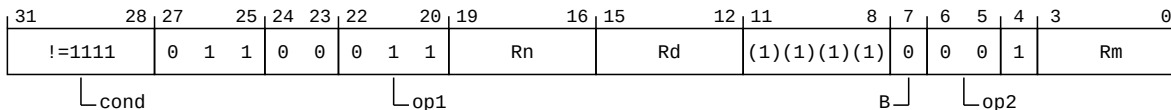
```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEventLocal();
end;
```

C5.1.182 SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



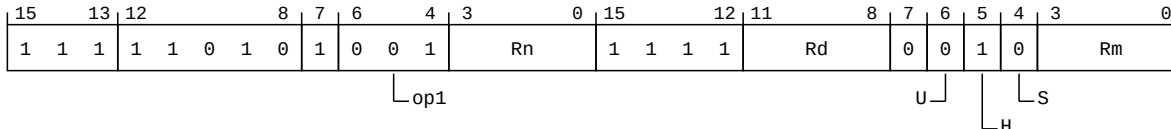
Encoding

SHADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SHADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

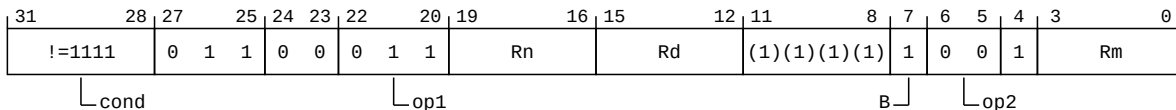
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[15:0]) + SInt(R(m)[15:0]);
    let sum2 : integer = SInt(R(n)[31:16]) + SInt(R(m)[31:16]);
    R(d)[15:0] = sum1[16:1];
    R(d)[31:16] = sum2[16:1];
end;
```

C5.1.183 SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



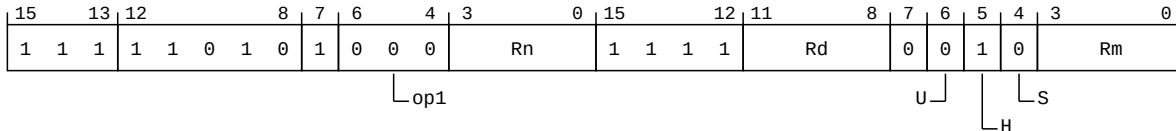
Encoding

SHADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SHADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

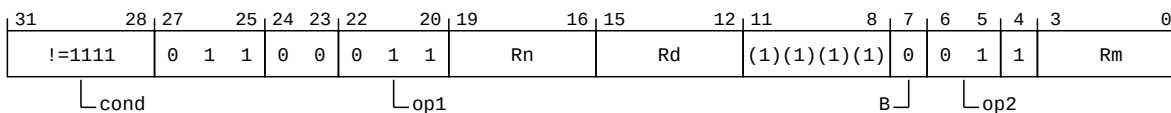
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = SInt(R(n)[7:0]) + SInt(R(m)[7:0]);
    let sum2 : integer = SInt(R(n)[15:8]) + SInt(R(m)[15:8]);
    let sum3 : integer = SInt(R(n)[23:16]) + SInt(R(m)[23:16]);
    let sum4 : integer = SInt(R(n)[31:24]) + SInt(R(m)[31:24]);
    R(d)[7:0]    = sum1[8:1];
    R(d)[15:8]   = sum2[8:1];
    R(d)[23:16]  = sum3[8:1];
    R(d)[31:24]  = sum4[8:1];
end;
```

C5.1.184 SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

SHASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

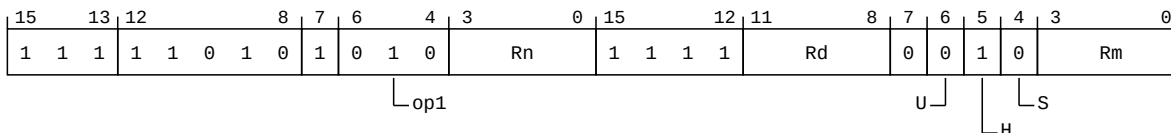
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

SHASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

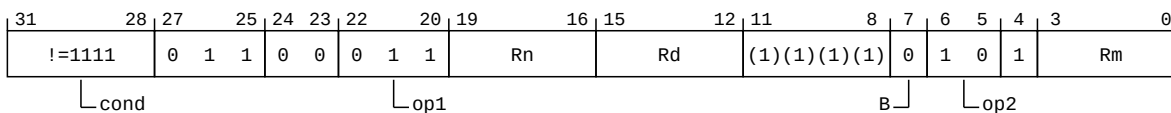
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = SInt(R(n)[15:0]) - SInt(R(m)[31:16]);
    let sum : integer = SInt(R(n)[31:16]) + SInt(R(m)[15:0]);
    R(d)[15:0] = diff[16:1];
    R(d)[31:16] = sum[16:1];
end;
```

C5.1.185 SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



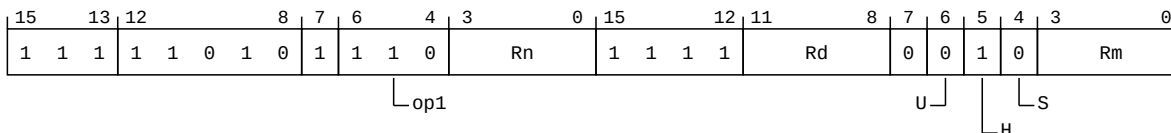
Encoding

SHSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SHSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

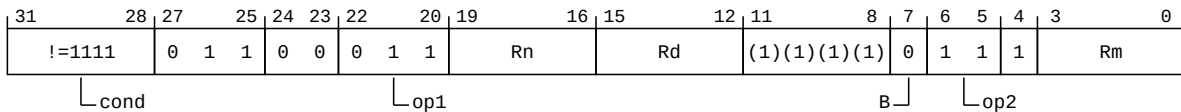
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0] = sum[16:1];
    R(d)[31:16] = diff[16:1];
end;
```

C5.1.186 SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



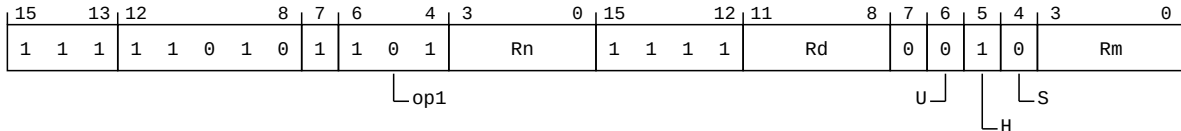
Encoding

SHSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SHSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

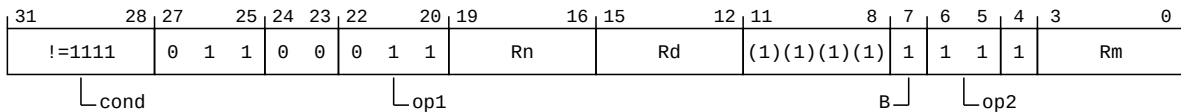
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0] = diff1[16:1];
    R(d)[31:16] = diff2[16:1];
end;
```

C5.1.187 SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



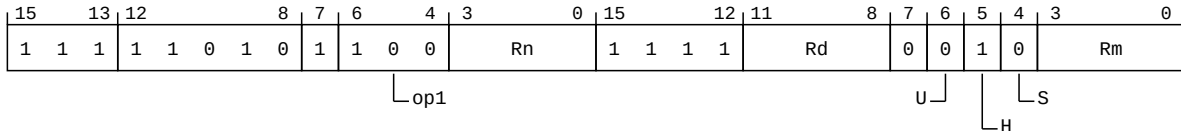
Encoding

SHSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SHSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0] = diff1[8:1];
    R(d)[15:8] = diff2[8:1];
    R(d)[23:16] = diff3[8:1];
    R(d)[31:24] = diff4[8:1];
end;
```

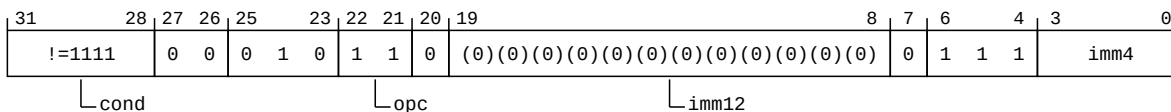
C5.1.188 SMC

Secure Monitor Call

This instruction is UNDEFINED.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



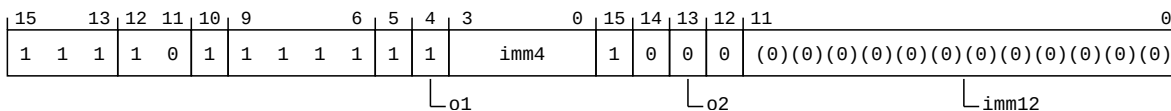
Encoding

$$\text{SMC}\{\langle c \rangle\}\{\langle \alpha \rangle\}\{\#\}\langle \text{imm4} \rangle$$

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
```

T1



Encoding

SMC { <c> } { <q> } { # } <imm4>

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<imm4>

Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the “imm4” field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but Arm does not recommend this.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    AArch32_CheckForSMCUndefOrTrap();

    Undefined();

end;
```

CONSTRAINED UNPREDICTABLE behavior

If `SCR.SCD == '1' && CurrentSecurityState() == SS_Secure`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

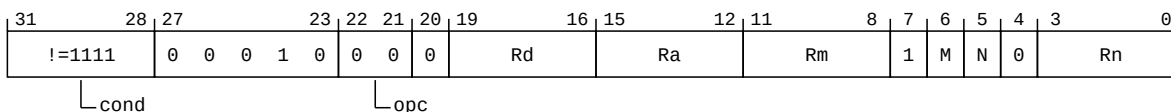
C5.1.189 **SMLABB, SMLABT, SMLATB, SMLATT**

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets **PSTATE.Q** to 1. It is not possible for overflow to occur during the multiplication.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMLABB variant

Applies when (M == 0 && N == 0)

SMLABB{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLABT variant

Applies when (M == 1 && N == 0)

SMLABT{<c>}{<q>}<Rd>,<Rn>,<Rm>,<Ra>

Encoding for the SMLATB variant

Applies when (M == 0 && N == 1)

SMLATB{<c>}{<q>}<Rd>,<Rn>,<Rm>,<Ra>

Encoding for the SMLATT variant

Applies when (M == 1 && N == 1)

SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```

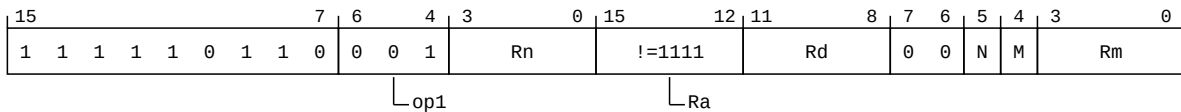
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);

let n_index : integer = UInt(N);
let m_index : integer = UInt(M);

if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;

```


T1



Encoding for the SMLABB variant

Applies when (N == 0 && M == 0)

SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLABT variant

Applies when (N == 0 && M == 1)

SMLABT{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLATB variant

Applies when (N == 1 && M == 0)

SMLATB{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLATT variant

Applies when (N == 1 && M == 1)

SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```

if Ra == 'l1l1' then See("SMULBB, SMULBT, SMULTB, SMULTT"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*:16];
    let operand2 : bits(16) = R(m)[m_index*:16];
    let result : integer = SInt(operand1) * SInt(operand2) + SInt(R(a));
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

C5.1.190 SMLAD, SMLADX

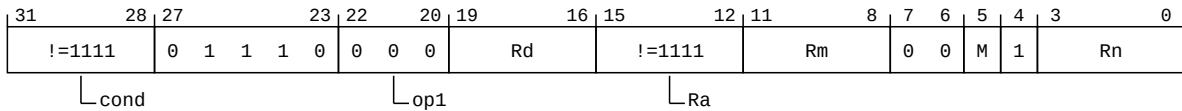
Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets `PSTATE.Q` to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMLAD variant

Applies when $(M == 0)$

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLADX variant

Applies when (M == 1)

SMLADX{<c>}{<q>}<Rd>, <Rn>, <Rm>, <Ra>

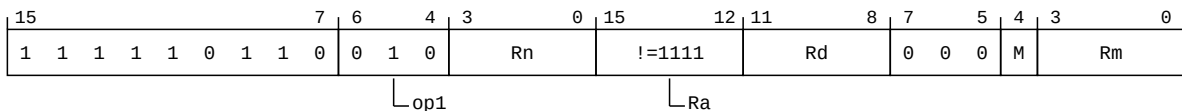
Decode for all variants of this encoding

```

if Ra == '1111' then See("SMUAD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the SMLAD variant

Applies when (M == 0)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLADX variant

Applies when (M == 1)

SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUAD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2 + SInt(R(a));
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

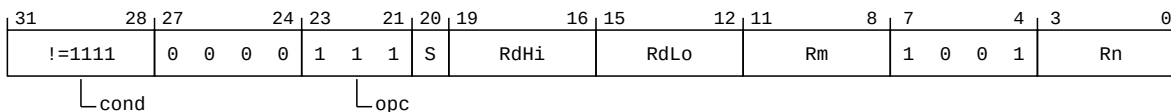
C5.1.191 SMLAL, SMLALS

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Flag setting variant

Applies when (S == 1)

SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the Not flag setting variant

Applies when $(S == 0)$

SMLAL{<c>}{<q>}<RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```

let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;

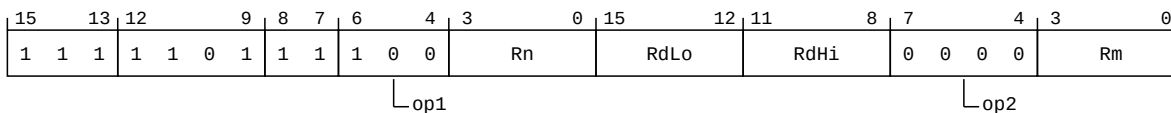
```

CONSTRAINED UNPREDICTABLE behavior

If $d_{Hi} == d_{Lo}$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

T1



Encoding

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = SInt(R(n)) * SInt(R(m)) + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

C5.1.192 SMLALBB, SMLALBT, SMLALTB, SMLALTT

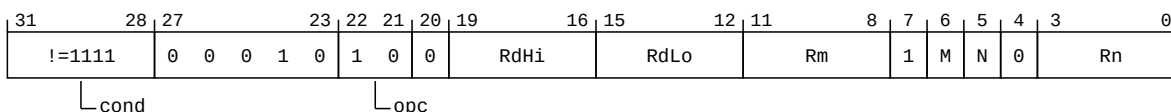
Signed Multiply Accumulate Long (halfwords)

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64} .

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMLALBB variant

Applies when (M == 0 && N == 0)

SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALBT variant

Applies when (M == 1 && N == 0)

SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALTB variant

Applies when (M == 0 && N == 1)

SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALTT variant

Applies when (M == 1 && N == 1)

SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

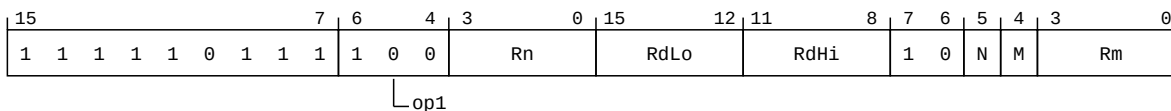
```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $dHi == dLo$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding for the SMLALBB variant

Applies when $(N == 0 \ \&\& \ M == 0)$

SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALBT variant

Applies when $(N == 0 \ \&\& \ M == 1)$

SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALTB variant

Applies when $(N == 1 \ \&\& \ M == 0)$

SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALTT variant

Applies when $(N == 1 \ \&\& \ M == 1)$

SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $dHi == dLo$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

For the “A1 SMLALBB”, “A1 SMLALBT”, “A1 SMLALTB”, and “A1 SMLALTT” variants: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the “Rn” field.

For the “T1 SMLALBB”, “T1 SMLALBT”, “T1 SMLALTB”, and “T1 SMLALTT” variants: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the “Rn” field.

<Rm>

For the “A1 SMLALBB”, “A1 SMLALBT”, “A1 SMLALTB”, and “A1 SMLALTT” variants: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the “Rm” field.

For the “T1 SMLALBB”, “T1 SMLALBT”, “T1 SMLALTB”, and “T1 SMLALTT” variants: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*16];
    let operand2 : bits(16) = R(m)[m_index*16];
    let result : integer = SInt(operand1) * SInt(operand2) + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

C5.1.193 SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual

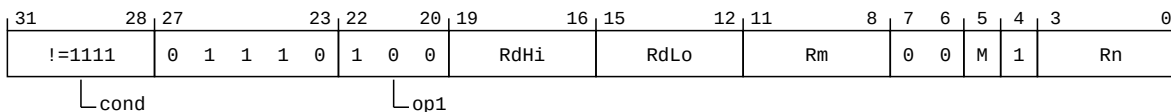
Signed Multiply Accumulate Long Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64} .

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the SMLALD variant

Applies when (M == 0)

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALDX variant

Applies when (M == 1)

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

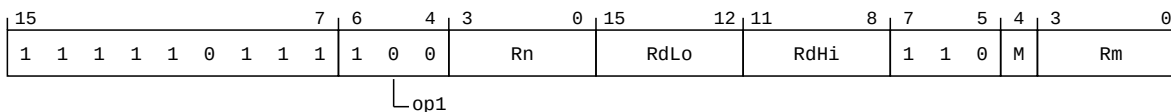
```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding for the SMLALD variant

Applies when (M == 0)

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLALDX variant

Applies when (M == 1)

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2 + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

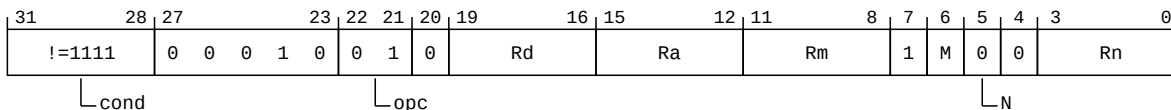
C5.1.194 SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets [PSTATE.Q](#) to 1. No overflow can occur during the multiplication.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMLAWB variant

Applies when (M == 0)

SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLAWT variant

Applies when (M == 1)

SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

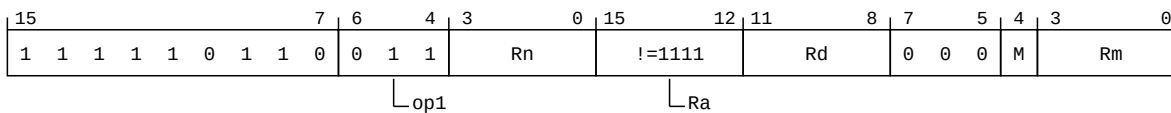
Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_high : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the SMLAWB variant

Applies when (M == 0)

SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLAWT variant

Applies when (M == 1)

SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMULWB, SMULWT"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_high : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
let operand2 : bits(16) = if m_high then R(m)[31:16] else R(m)[15:0];
let result : integer = SInt(R(n)) * SInt(operand2) + (SInt(R(a)) << 16);
R(d) = result[47:16];
if (result >> 16) != SInt(R(d)) then // Signed overflow
    PSTATE.Q = '1';
end;
end;
```

C5.1.195 SMLSD, SMLSDX

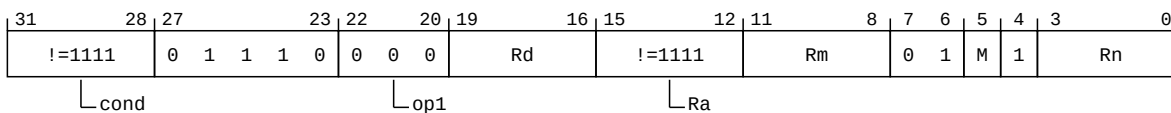
Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets [PSTATE.Q](#) to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMLSD variant

Applies when (M == 0)

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLSDX variant

Applies when (M == 1)

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

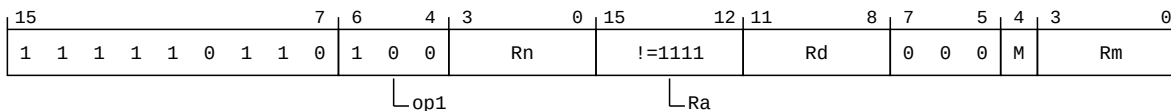
Decode for all variants of this encoding

```

if Ra == '1111' then See("SMUSD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the SMLSD variant

Applies when (M == 0)

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMLSDX variant

Applies when (M == 1)

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMUSD"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
let result : integer = (product1 - product2) + SInt(R(a));
R(d) = result[31:0];
if result != SInt(result[31:0]) then // Signed overflow
    PSTATE.Q = '1';
end;
end;
```

C5.1.196 SMLS LD, SMLS LD X

Signed Multiply Subtract Long Dual

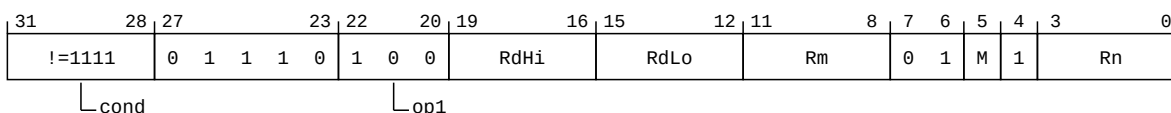
Signed Multiply Subtract Long Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64} .

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the SMLS LD variant

Applies when (M == 0)

SMLS LD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLS LD X variant

Applies when (M == 1)

SMLS LD X{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

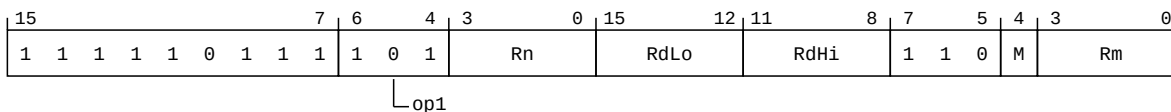
```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding for the SMLS LD variant

Applies when (M == 0)

SMLS LD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the SMLS LD X variant

Applies when (M == 1)

SMLS LD X{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If dHi == dLo, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = (product1 - product2) + SInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
end;
```

C5.1.197 SMMLA, SMMLAR

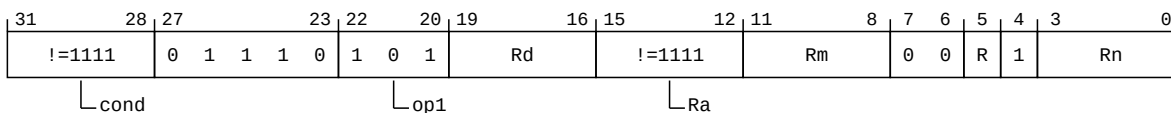
Signed Most Significant Word Multiply Accumulate

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMMLA variant

Applies when $(R == 0)$

SMMLA { <c> } { <q> } <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMMLAR variant

Applies when $(R == 1)$

SMMLAR{<c>}{<q>}<Rd>,<Rn>,<Rm>,<Ra>

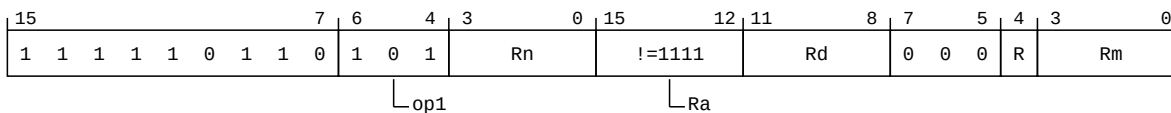
Decode for all variants of this encoding

```

if Ra == '1111' then See("SMMUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the SMMLA variant

Applies when ($R = 0$)

$$\text{SMMLA} \{ \langle c \rangle \} \{ \langle q \rangle \} \quad \langle Rd \rangle, \quad \langle Rn \rangle, \quad \langle Rm \rangle, \quad \langle Ra \rangle$$

Encoding for the SMMLAR variant

Applies when (R == 1)

SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Decode for all variants of this encoding

```
if Ra == '1111' then See("SMMUL"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
var result : integer = (SInt(R(a)) << 32) + SInt(R(n)) * SInt(R(m));
if round then result = result + 0x80000000; end;
R(d) = result[63:32];
end;
```

C5.1.198 SMMLS, SMMLSR

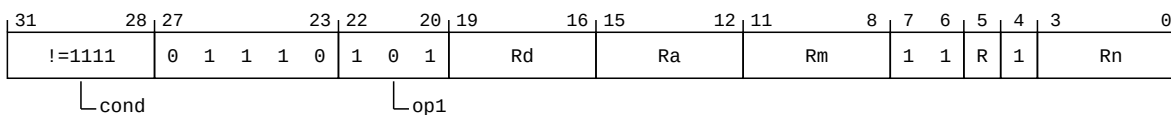
Signed Most Significant Word Multiply Subtract

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the result of the subtraction before the high word is extracted.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMMLS variant

Applies when $(R == 0)$

SMMLS { <c> } { <q> } <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMMLSR variant

Applies when $(R == 1)$

$$\text{SMMLSR}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle Rd \rangle, \langle Rn \rangle, \langle Rm \rangle, \langle Ra \rangle$$

Decode for all variants of this encoding

```

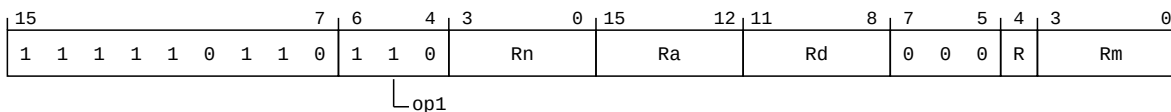
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);

let round : boolean = (R == '1');

if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the SMMLS variant

Applies when $(R == 0)$

SMMLS {<c>} {<q>} <Rd>, <Rn>, <Rm>, <Ra>

Encoding for the SMMLSR variant

Applies when $(R == 1)$

$$\text{SMMLSR}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle Rd \rangle, \langle Rn \rangle, \langle Rm \rangle, \langle Ra \rangle$$

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
var result : integer = (SInt(R(a)) << 32) - SInt(R(n)) * SInt(R(m));
if round then result = result + 0x80000000; end;
R(d) = result[63:32];
end;
```

C5.1.199 SMMUL, SMMULR

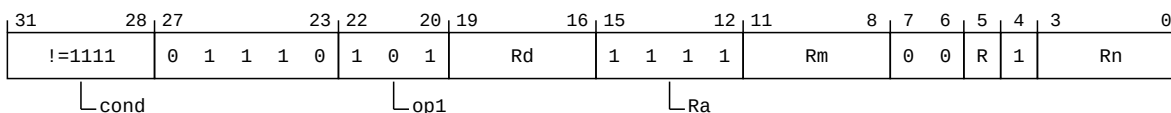
Signed Most Significant Word Multiply

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0×80000000 is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the SMMUL variant

Applies when (R == 0)

SMMUL{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMMULR variant

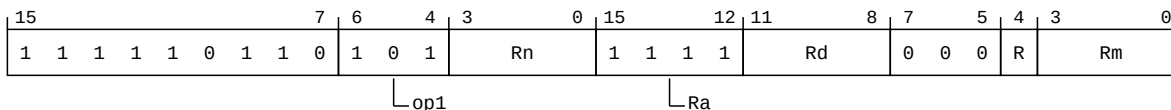
Applies when (R == 1)

SMMULR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let round : boolean = (R == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the SMMUL variant

Applies when (R == 0)

SMMUL{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMMULR variant

Applies when (R == 1)

SMMULR{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let round : boolean = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var result : integer = SInt(R(n)) * SInt(R(m));
    if round then result = result + 0x80000000; end;
    R(d) = result[63:32];
end;
```

C5.1.200 SMUAD, SMUADX

Signed Dual Multiply Add

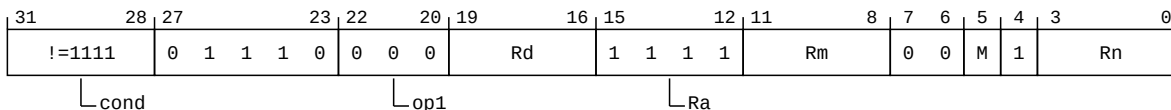
Signed Dual Multiply Add performs two signed 16 x 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets [PSTATE.Q](#) to 1 if the addition overflows. The multiplications cannot overflow.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMUAD variant

Applies when (M == 0)

SMUAD{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMUADX variant

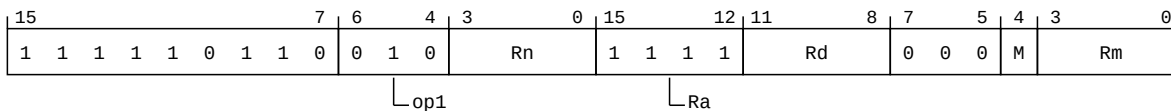
Applies when (M == 1)

SMUADX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the SMUAD variant

Applies when (M == 0)

SMUAD{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMUADX variant

Applies when (M == 1)

SMUADX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 + product2;
    R(d) = result[31:0];
    if result != SInt(result[31:0]) then // Signed overflow
        PSTATE.Q = '1';
    end;
end;
```

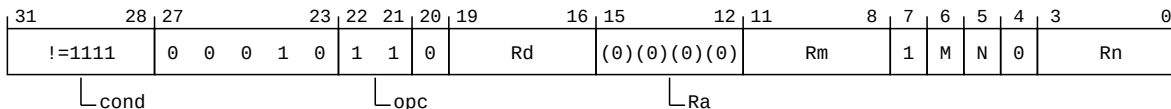
C5.1.201 SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords)

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMULBB variant

Applies when (M == 0 && N == 0)

SMULBB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULBT variant

Applies when (M == 1 && N == 0)

SMULBT{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULTB variant

Applies when (M == 0 && N == 1)

SMULTB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULTT variant

Applies when (M == 1 && N == 1)

SMULTT{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

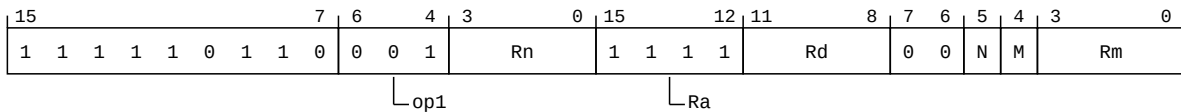
Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the SMULBB variant

Applies when (N == 0 && M == 0)

SMULBB{<c>}{<q>}{<Rd>,<Rn>,<Rm>}

Encoding for the SMULBT variant

Applies when (N == 0 && M == 1)

SMULBT{<c>}{<q>}{<Rd>,<Rn>,<Rm>}

Encoding for the SMULTB variant

Applies when (N == 1 && M == 0)

SMULTB{<c>}{<q>}{<Rd>,<Rn>,<Rm>}

Encoding for the SMULTT variant

Applies when (N == 1 && M == 1)

SMULTT{<c>}{<q>}{<Rd>,<Rn>,<Rm>}

Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let n_index : integer = UInt(N);
let m_index : integer = UInt(M);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 : bits(16) = R(n)[n_index*:16];
    let operand2 : bits(16) = R(m)[m_index*:16];
    let result : integer = SInt(operand1) * SInt(operand2);
    R(d) = result[31:0];
    // Signed overflow cannot occur
end;
```

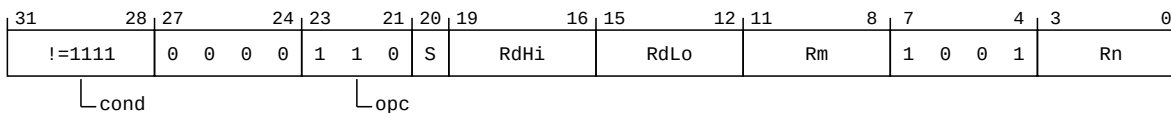
C5.1.202 SMULL, SMULLS

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Flag setting variant

Applies when (S == 1)

```
SMULLS{<c>}{<q>}    <RdLo>, <RdHi>, <Rn>, <Rm>
```

Encoding for the Not flag setting variant

Applies when $(S == 0)$

```
SMULL{<c>}{<q>}    <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```

let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;

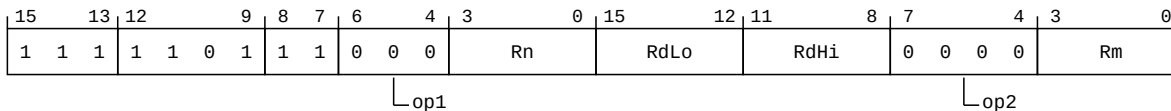
```

CONSTRAINED UNPREDICTABLE behavior

If $d_{Hi} == d_{Lo}$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

T1



Encoding

SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<RdLo>

Is the general-purpose destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = SInt(R(n)) * SInt(R(m));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

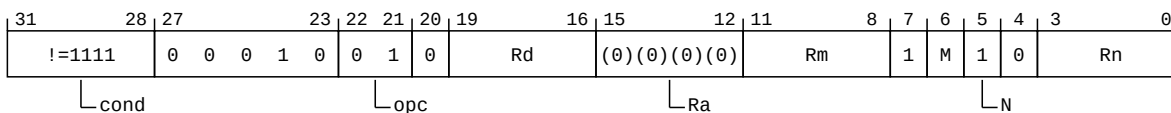

C5.1.203 SMULWB, SMULWT

Signed Multiply (word by halfword)

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the SMULWB variant

Applies when (M == 0)

SMULWB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULWT variant

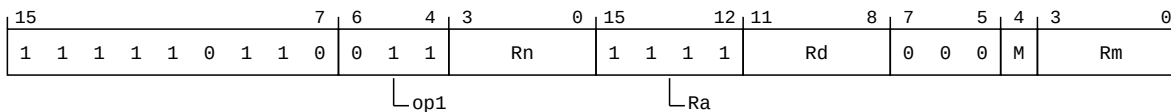
Applies when (M == 1)

SMULWT{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_high : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the SMULWB variant

Applies when (M == 0)

SMULWB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMULWT variant

Applies when (M == 1)

SMULWT{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_high : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(16) = if m_high then R(m)[31:16] else R(m)[15:0];
    let product : integer = SInt(R(n)) * SInt(operand2);
    R(d) = product[47:16];
    // Signed overflow cannot occur
end;
```

C5.1.204 SMUSD, SMUSDX

Signed Multiply Subtract Dual

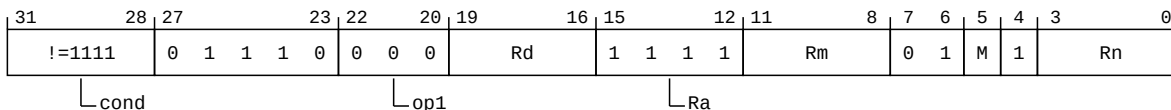
Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow cannot occur.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SMUSD variant

Applies when (M == 0)

SMUSD{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMUSDX variant

Applies when (M == 1)

SMUSDX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

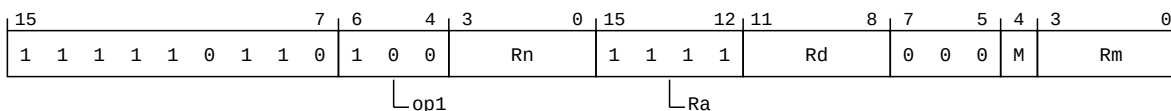
Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the SMUSD variant

Applies when (M == 0)

SMUSD{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Encoding for the SMUSDX variant

Applies when (M == 1)

SMUSDX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let m_swap : boolean = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand2 : bits(32) = if m_swap then ROR(R(m),16) else R(m);
    let product1 : integer = SInt(R(n)[15:0]) * SInt(operand2[15:0]);
    let product2 : integer = SInt(R(n)[31:16]) * SInt(operand2[31:16]);
    let result : integer = product1 - product2;
    R(d) = result[31:0];
    // Signed overflow cannot occur
end;
```

C5.1.205 SRS, SRSDA, SRSDB, SRSIA, SRSIB

Store Return State stores the LR_<current_mode> and SPSR_<current_mode> to the stack of a specified mode. For information about memory accesses see [Memory accesses](#).

SRS is UNDEFINED in Hyp mode.

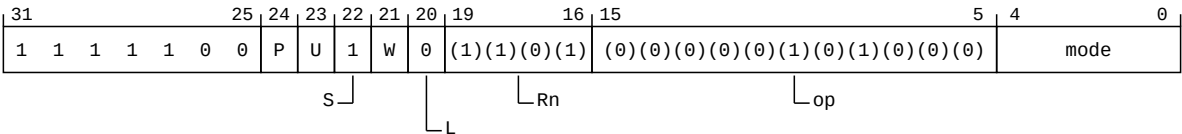
SRS is CONSTRAINED UNPREDICTABLE if it is executed in User or System mode, or if the specified mode is any of the following:

- Not implemented.
- A mode that [AArch32 PE modes](#) does not show.
- Hyp mode.
- Monitor mode, if the SRS instruction is executed in Non-secure state.

If EL3 is using AArch64 and an SRS instruction that is executed in a Secure EL1 mode specifies Monitor mode, it is trapped to EL3.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Decrement After variant

Applies when (P == 0 && U == 0)

SRSDA{<c>}{<q>} SP{!}, #<mode>

Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0)

SRSDB{<c>}{<q>} SP{!}, #<mode>

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

Encoding for the Increment Before variant

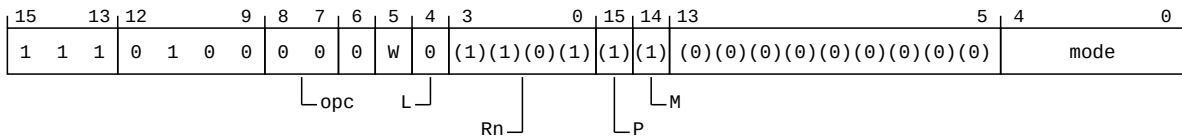
Applies when (P == 1 && U == 1)

SRSIB{<c>}{<q>} SP{!}, #<mode>

Decode for all variants of this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
```

T1



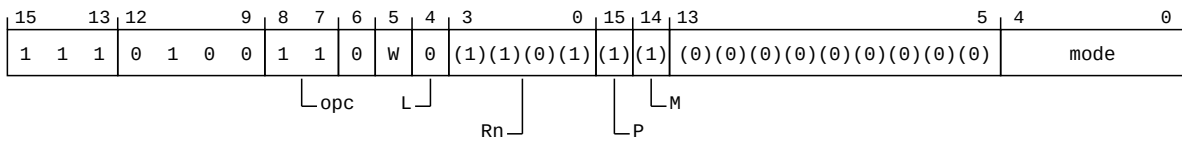
Encoding

SRSDB{<c>}{<q>} SP{!}, #<mode>

Decode for this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = FALSE;
let wordhigher : boolean = FALSE;
```

T2



Encoding

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

Decode for this encoding

```
let wback : boolean = (W == '1');
let increment : boolean = TRUE;
let wordhigher : boolean = FALSE;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [SRS \(T32\)](#) and [SRS \(A32\)](#).

Assembler Symbols

 $\langle c \rangle$

For the “Decrement After”, “Decrement Before”, “Increment After”, and “Increment Before” variants: see [Standard assembler syntax fields](#). `<C>` must be `ALL` or omitted.

For the “T1” and “T2” variants: see [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<mode>

Is the number of the mode whose Banked SP is used as the base register, encoded in the “mode” field. For details of PE modes and their numbers see [AArch32 state PE modes](#).

IA

For the “Increment After” variant: is an optional suffix to indicate the Increment After variant.

For the “T2” variant: is an optional suffix for the Increment After form.

SRSFA, SRSEA, SRSFD, and SRSED are pseudo-instructions for SRSIB, SRSIA, SRSDB, and SRSDA respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```

if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then           // UNDEFINED at EL2
            Undefined();
        end;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        elsif mode == M32_Hyp then          // Check for attempt to access Hyp mode SP
            UnpredictableProcedure();
        elsif mode == M32_Monitor then      // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || CurrentSecurityState() != SS_Secure then
                UnpredictableProcedure();
            end;
        elsif BadMode(mode) then
            UnpredictableProcedure();
        end;

        let base : bits(32) = Rmode(13,mode);
        var address : bits(32) = if increment then base else base-8;
        if wordhigher then address = address+4; end;
        MemA{32}(address) = LR();
        MemA{32}(address+4) = SPSR_curr();
        if wback then Rmode(13,mode) = if increment then base+8 else base-8; end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then           // UNDEFINED at EL2
            Undefined();
        end;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UnpredictableProcedure();
        elsif mode == M32_Hyp then          // Check for attempt to access Hyp mode SP
            UnpredictableProcedure();
        elsif mode == M32_Monitor then      // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || CurrentSecurityState() != SS_Secure then
                UnpredictableProcedure();
            end;
        elsif BadMode(mode) then
            UnpredictableProcedure();
        end;

        let base : bits(32) = Rmode(13,mode);

```

```
var address : bits(32) = if increment then base else base-8;  
if wordhigher then address = address+4; end;  
MemA{32}(address) = LR();  
MemA{32}(address+4) = SPSR_curr();  
if wback then Rmode(13,mode) = if increment then base+8 else base-8; end;  
end;  
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M` IN {`M32_User`,`M32_System`}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode == M32_Hyp`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode == M32_Monitor` && (!`HaveEL(EL3)` || `CurrentSecurityState()` != `SS_Secure`), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `BadMode(mode)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction stores to the stack of the mode in which it is executed.
- The instruction stores to an UNKNOWN address, and if the instruction specifies writeback then any general-purpose register that can be accessed from the current Exception level without a privilege violation becomes UNKNOWN.

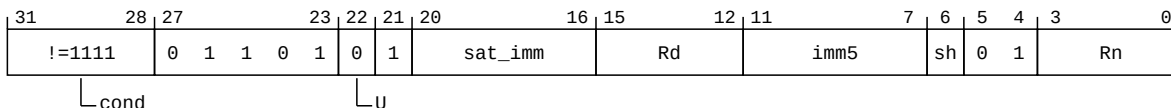
C5.1.206 SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

This instruction sets [PSTATE.Q](#) to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Arithmetic shift right variant

Applies when (sh == 1)

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Encoding for the Logical shift left variant

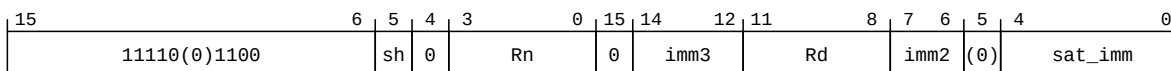
Applies when (sh == 0)

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(sh::'0', imm5);
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the Arithmetic shift right variant

Applies when (sh == 1 && !(imm3 == 000 && imm2 == 00))

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Encoding for the Logical shift left variant

Applies when (sh == 0)

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

Decode for all variants of this encoding

```

if sh == '1' && (imm3::imm2) == '00000' then See("SSAT16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(sh::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the bit position for saturation, in the range 1 to 32, encoded in the “sat_imm” field as <imm>-1.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<amount>

For the “A1 Arithmetic shift right” variant: is the shift amount, in the range 1 to 32 encoded in the “imm5” field as <amount> modulo 32.

For the “A1 Logical shift left” variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.

For the “T1 Arithmetic shift right” variant: is the shift amount, in the range 1 to 31 encoded in the “imm3:imm2” field as <amount>.

For the “T1 Logical shift left” variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm3:imm2” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let operand : bits(32) = Shift{}(R(n), shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
    let (result, sat) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(operand));
    R(d) = SignExtend{32}(result);
    if sat then
        PSTATE.Q = '1';
    end;
end;

```

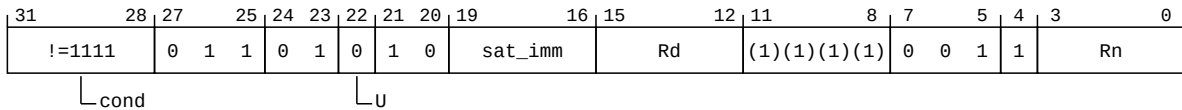
C5.1.207 SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

This instruction sets **PSTATE.Q** to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

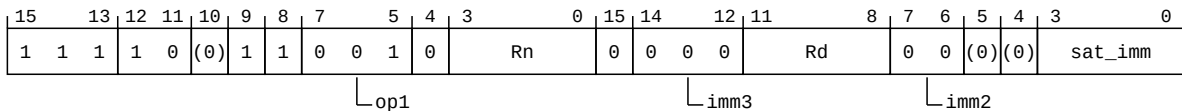
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm)+1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;

```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the bit position for saturation, in the range 1 to 16, encoded in the “sat_imm” field as <imm>-1.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

Operation

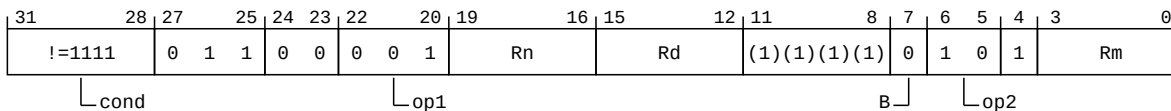
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result1, sat1) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(R(n)[15:0]));
    let (result2, sat2) : (bits(saturate_to), boolean) = SignedSatQ{saturate_to}(SInt(R(n)[31:16]));
    R(d)[15:0] = SignExtend{16}(result1);
    R(d)[31:16] = SignExtend{16}(result2);
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```

C5.1.208 SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



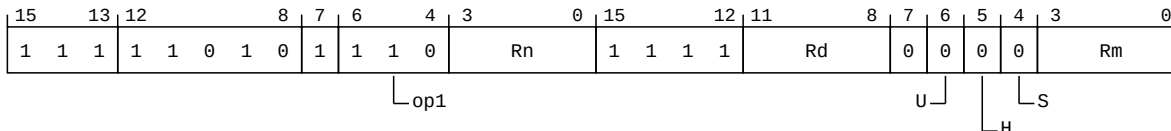
Encoding

SSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

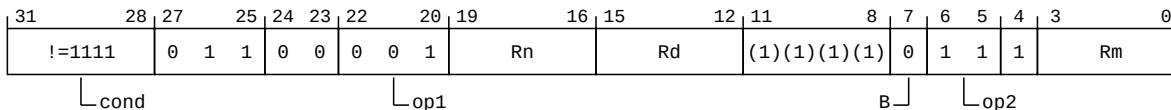
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = SInt(R(n)[15:0]) + SInt(R(m)[31:16]);
    let diff : integer = SInt(R(n)[31:16]) - SInt(R(m)[15:0]);
    R(d)[15:0] = sum[15:0];
    R(d)[31:16] = diff[15:0];
    PSTATE.GE[1:0] = if sum >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff >= 0 then '11' else '00';
end;
```

C5.1.209 SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets **PSTATE**.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



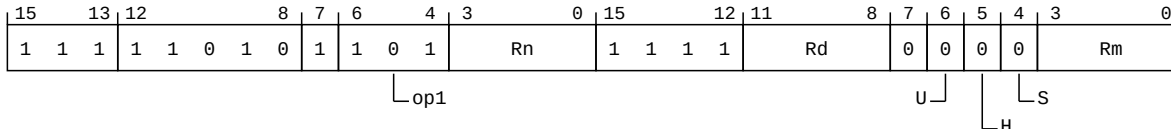
Encoding

SSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

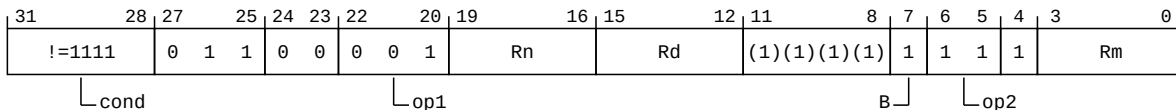
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[15:0]) - SInt(R(m)[15:0]);
    let diff2 : integer = SInt(R(n)[31:16]) - SInt(R(m)[31:16]);
    R(d)[15:0] = diff1[15:0];
    R(d)[31:16] = diff2[15:0];
    PSTATE.GE[1:0] = if diff1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff2 >= 0 then '11' else '00';
end;
```

C5.1.210 SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results of the subtractions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



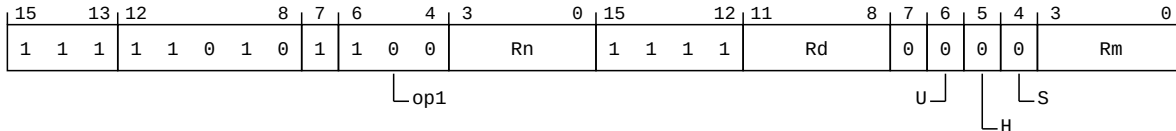
Encoding

SSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = SInt(R(n)[7:0]) - SInt(R(m)[7:0]);
    let diff2 : integer = SInt(R(n)[15:8]) - SInt(R(m)[15:8]);
    let diff3 : integer = SInt(R(n)[23:16]) - SInt(R(m)[23:16]);
    let diff4 : integer = SInt(R(n)[31:24]) - SInt(R(m)[31:24]);
    R(d)[7:0] = diff1[7:0];
    R(d)[15:8] = diff2[7:0];
    R(d)[23:16] = diff3[7:0];
    R(d)[31:24] = diff4[7:0];
    PSTATE.GE[0] = if diff1 >= 0 then '1' else '0';
    PSTATE.GE[1] = if diff2 >= 0 then '1' else '0';
    PSTATE.GE[2] = if diff3 >= 0 then '1' else '0';
    PSTATE.GE[3] = if diff4 >= 0 then '1' else '0';
end;
```

C5.1.211 STC

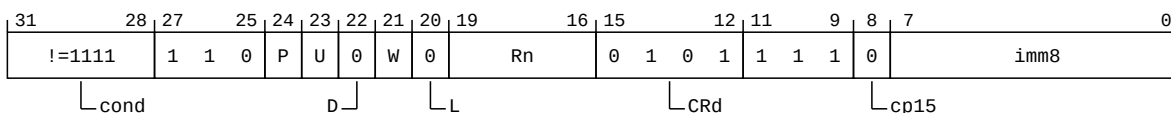
Store data to System register calculates an address from a base register value and an immediate offset, and stores a word from the [DBGDTRRXint](#) System register to memory. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses, see [Memory accesses](#).

In an implementation that includes EL2, the permitted `STC` access to [DBGDTRRXint](#) can be trapped to Hyp mode, meaning that an attempt to execute an `STC` instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [HDCR.TDA](#).

For simplicity, the `STC` pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STC{<c>}{<q>} p14, c5, [<Rn>{, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

STC{<c>}{<q>} p14, c5, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STC{<c>}{<q>} p14, c5, [<Rn>, #<+/-><imm>]!

Encoding for the Unindexed variant

Applies when (P == 0 && U == 1 && W == 0)

STC{<c>}{<q>} p14, c5, [<Rn>], <option>

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;

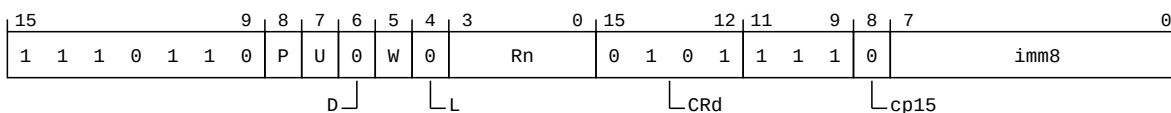
```

CONSTRAINED UNPREDICTABLE behavior

If $n == 15$ && $wback$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

T1



Encoding for the Offset variant

Applies when $(P == 1 \ \&\& \ W == 0)$

STC{<c>}{<q>} p14, c5, [<Rn>{, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when $(P == 0 \ \&\& \ W == 1)$

STC{<c>}{<q>} p14, c5, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when $(P == 1 \ \&\& \ W == 1)$

STC{<c>}{<q>} p14, c5, [<Rn>, #<+/-><imm>]!

Encoding for the Unindexed variant

Applies when $(P == 0 \ \&\& \ U == 1 \ \&\& \ W == 0)$

STC{<c>}{<q>} p14, c5, [<Rn>], <option>

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then Undefined(); end;
let n : integer = UInt(Rn);
let cp : integer = 14;
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

`<c>`

See [Standard assembler syntax fields](#).

`<q>`

See [Standard assembler syntax fields](#).

`<Rn>`

For the “A1 Offset” and “A1 Unindexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “A1 Post-indexed”, “A1 Pre-indexed”, “T1 Offset”, “T1 Post-indexed”, “T1 Pre-indexed”, and “T1 Unindexed” variants: is the general-purpose base register, encoded in the “Rn” field.

`+/-`

Specifies the offset is added to or subtracted from the base register, defaulting to `+` if omitted and encoded in “U”:

U	+/-
0	-
1	+

`<imm>`

Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the “imm8” field, as `<imm>/4`.

`<option>`

Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the “imm8” field. The value of this field is ignored when executing this instruction.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);

    // System register read from DBGDTRXint.
    AArch32_SysRegRead(cp, ThisInstr(), address[31:0]);
    if wback then R(n) = offset_addr; end;
end;
```

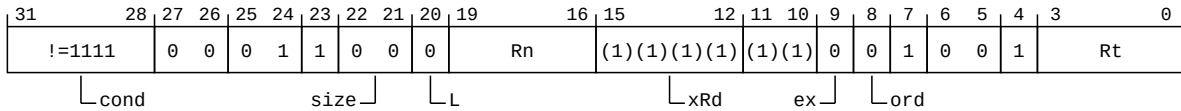
C5.1.212 STL

Store-Release Word stores a word from a register to memory. The instruction also has memory ordering semantics as described in [Load-Acquire](#), [Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



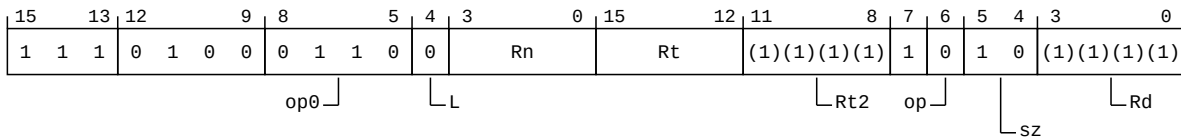
Encoding

STL{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

STL{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{32}(address) = R(t);
end;
```

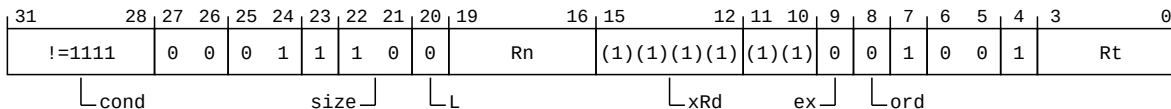
C5.1.213 STLB

Store-Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics as described in [Load-Acquire](#), [Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



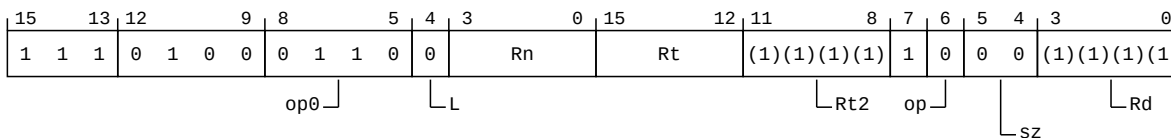
Encoding

STLB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

STLB{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{8}(address) = R(t)[7:0];
end;
```

C5.1.214 STLEX

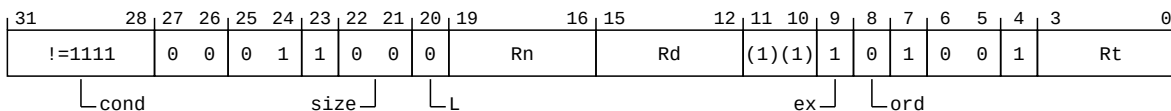
Store-Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

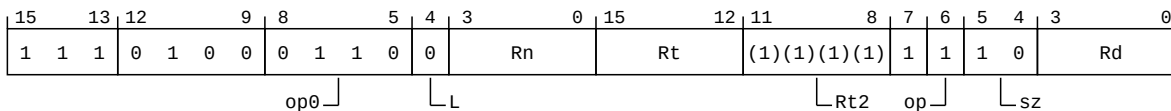
If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1



Encoding

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
let address : bits(32) = R(n);
if AArch32_ExclusiveMonitorsPass(address,4) then
    MemO{32}(address) = R(t);
    R(d) = ZeroExtend{32}('0');
else
    R(d) = ZeroExtend{32}('1');
end;
end;
```

C5.1.215 STLEXB

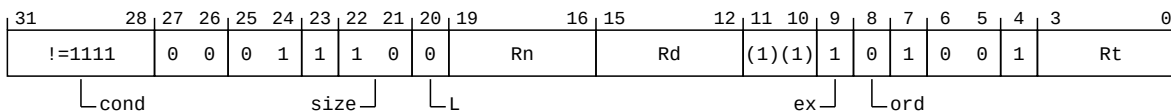
Store-Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

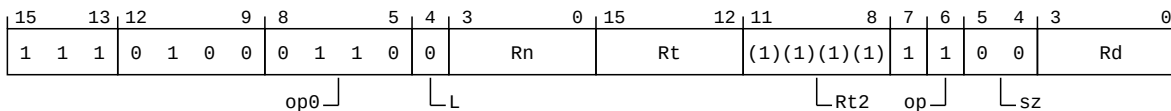
If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1



Encoding

STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);  
let t : integer = UInt(Rt);  
let n : integer = UInt(Rn);  
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;  
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
let address : bits(32) = R(n);
if AArch32_ExclusiveMonitorsPass(address,1) then
    MemO{8}(address) = R(t)[7:0];
    R(d) = ZeroExtend{32}('0');
else
    R(d) = ZeroExtend{32}('1');
end;
end;
```

C5.1.216 STLEXD

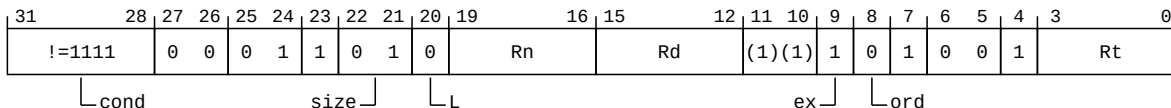
Store-Release Exclusive Doubleword stores a doubleword from two registers to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
if d == 15 || Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

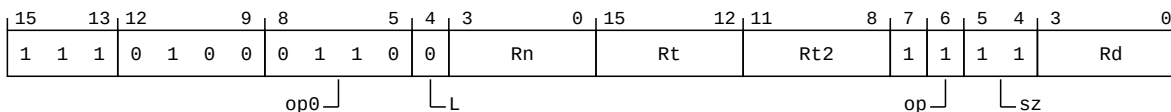
If $Rt<0> == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $Rt<0> = '0'$.
- The instruction executes with the additional decode: $t2 = t$.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If $Rt == '1110'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15.

T1



Encoding

STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. <Rt> must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    let value : bits(64) = if BigEndian(AccessType_GPR) then R(t)::R(t2) else R(t2)::R(t);
    if AArch32_ExclusiveMonitorsPass(address, 8) then
        MemO{64}(address) = value;
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;

```

C5.1.217 STLEXH

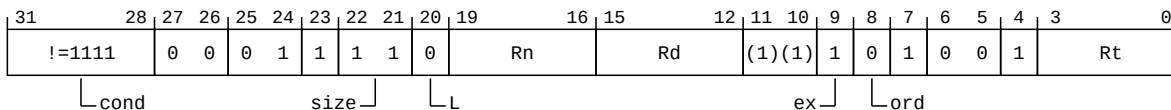
Store-Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

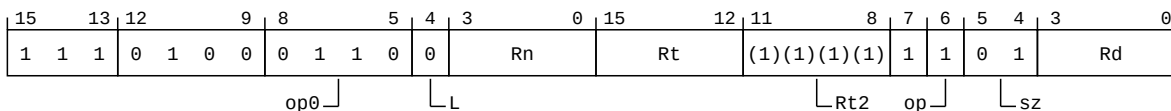
If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1



Encoding

STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
let address : bits(32) = R(n);
if AArch32_ExclusiveMonitorsPass(address,2) then
    MemO{16}(address) = R(t)[15:0];
    R(d) = ZeroExtend{32}('0');
else
    R(d) = ZeroExtend{32}('1');
end;
end;
```

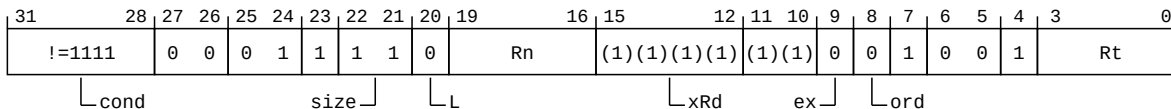
C5.1.218 STLH

Store-Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics as described in [Load-Acquire, Store-Release](#).

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



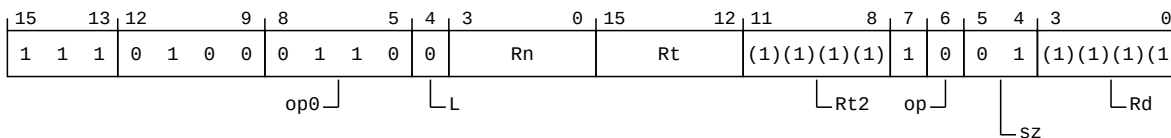
Encoding

STLH{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

STLH{<c>}{<q>} <Rt>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if t == 15 || n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);
    MemO{16}(address) = R(t)[15:0];
end;
```

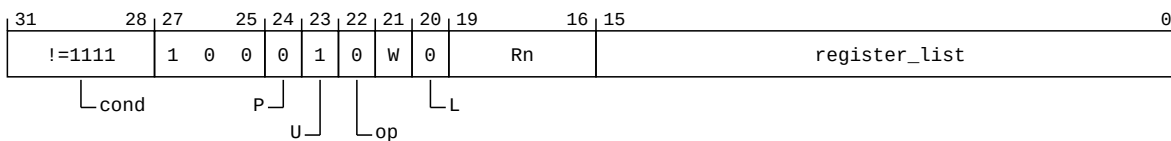
C5.1.219 STM, STMIA, STMEA

Store Multiple (Increment After, Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2).

A1



Encoding

```
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

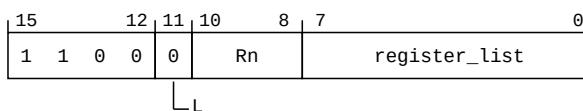
If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include `R15`. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15` && `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

T1



Encoding

STM{IA}{<c>}{<q>} <Rn>!, <registers> // (Preferred syntax)

STMEA{<c>}{<q>} <Rn>!, <registers> // (Alternate syntax, Empty Ascending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = '00000000'::register_list;
let wback : boolean = TRUE;
if BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

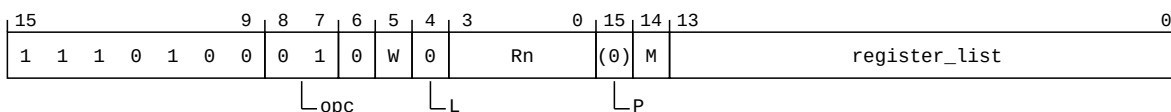
If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

T2



Encoding

STM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be
→ represented in T1)

STMEA{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack, if <Rn>, '!' and
→ <registers> can be represented in T1)

STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in Using R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

IA

Is an optional suffix for the Increment After form.

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

For the “A1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For the “T1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the “register_list” field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For the “T2” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain the LR. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n);
    let length : integer = 4*BitCount(registers);
    let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                // Only possible for encodings T1 and A1
                MemS{32}(address, within8byte) = ARBITRARY : bits(32);
            else
                MemS{32}(address, within8byte) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then // Only possible for encoding A1
        MemS{32}(address, within8byte) = PCStoreValue();
    end;
    if wback then R(n) = R(n) + 4*BitCount(registers); end;
end;

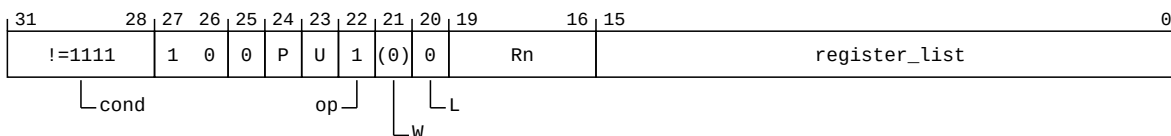
```

C5.1.220 STM (User registers)

Store Multiple (User registers)

In an EL1 mode other than System mode, Store Multiple (User registers) stores multiple User mode registers to consecutive memory locations using an address from a base register. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Store Multiple (User registers) is UNDEFINED in Hyp mode, and CONSTRAINED UNPREDICTABLE in User or System modes.



Encoding

STM{<amode>}{<c>}{<q>} <Rn>, <registers>^

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let increment : boolean = (U == '1');
let wordhigher : boolean = (P == U);
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If BitCount(registers) < 1, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<amode>

is one of:

- DA** Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
- ED** Empty Descending. For this instruction, a synonym for DA.
- DB** Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
- FD** Full Descending. For this instruction, a synonym for DB.
- IA** Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
- EA** Empty Ascending. For this instruction, a synonym for IA.
- IB** Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.
- FA** Full Ascending. For this instruction, a synonym for IB.

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<registers>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the STM instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        Undefined();
    elseif PSTATE.M IN {M32_User,M32_System} then
        UnpredictableProcedure();
    else
        let length : integer = 4*BitCount(registers);
        var address : bits(32) = if increment then R(n) else R(n)-length;
        let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
        if wordhigher then address = address+4; end;
        for i = 0 to 14 do
            if registers[i] == '1' then // Store User mode register
                MemS{32}(address, within8byte) = Rmode(i, M32_User);
                address = address + 4;
            end;
        end;
        if registers[15] == '1' then
            MemS{32}(address, within8byte) = PCStoreValue();
        end;
    end;
end;

```

CONSTRAINED UNPREDICTABLE behavior

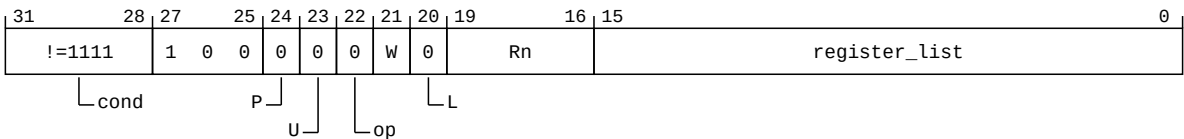
If PSTATE.M IN {M32_User,M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

C5.1.221 STMDA, STMED

Store Multiple Decrement After (Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).



Encoding

```
STMDA{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMED{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>
See [Standard assembler syntax fields](#).

<q>
See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = (R(n) - 4*BitCount(registers)) + 4;
    let length : integer = 4*BitCount(registers);
    let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address, within8byte) = ARBITRARY : bits(32);
            else
                MemS{32}(address, within8byte) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        MemS{32}(address, within8byte) = PCStoreValue();
    end;
    if wback then R(n) = R(n) - 4*BitCount(registers); end;
end;

```

C5.1.222 STMDB, STMFd

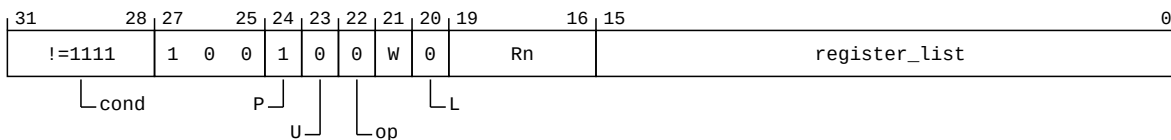
Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).

This instruction is used by the alias [PUSH \(multiple registers\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFd{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)

Decode for this encoding

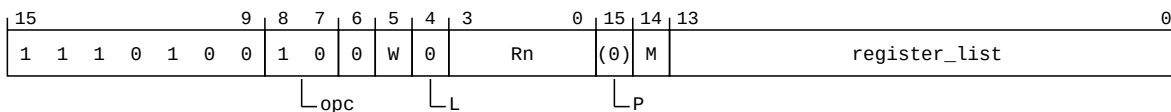
```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

T1



Encoding

STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFd{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = P::M::register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 2 then UnpredictableProcedure(); end;
if wback && registers[n] == '1' then UnpredictableProcedure(); end;
if registers[13] == '1' then UnpredictableProcedure(); end;
if registers[15] == '1' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

For the “A1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

For the “T1” variant: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The registers in the list must be in the range R0-R12, encoded in the “register_list” field, and can optionally contain the LR. If the LR is in the list, the “M” field is set to 1, otherwise it defaults to 0.

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (multiple registers)	A1	W == '1' && Rn == '1101' && BitCount(register_list) > 1
PUSH (multiple registers)	T1	W == '1' && Rn == '1101' && BitCount(M : register_list) > 1
PUSH (multiple registers)	T1	BitCount(M : register_list) > 1

Operation

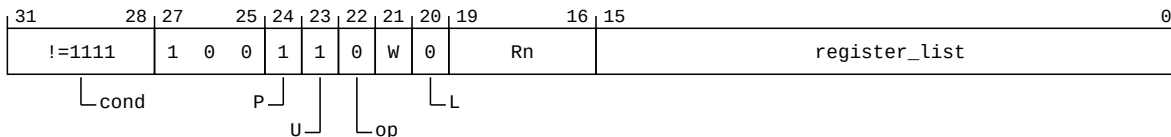
```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) - 4*BitCount(registers);
    let length : integer = 4*BitCount(registers);
    let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address, within8byte) = ARBITRARY : bits(32);
            else
                MemS{32}(address, within8byte) = R(i);
            end;
            address = address + 4;
        end;
    end;
```

```
    if registers[15] == '1' then // Only possible for encoding A1
        MemS{32}(address, within8byte) = PCStoreValue();
    end;
    if wback then R(n) = R(n) - 4*BitCount(registers); end;
end;
```

C5.1.223 STMIB, STMFA

Store Multiple Increment Before (Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also [Encoding of lists of general-purpose registers and the PC](#).



Encoding

```
STMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMFA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)
```

Decode for this encoding

```
let n : integer = UInt(Rn);
let registers : bits(16) = register_list;
let wback : boolean = (W == '1');
if n == 15 || BitCount(registers) < 1 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include `R15`. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15` && `wback`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

!

The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the “W” field as 1, otherwise this field defaults to 0.

<registers>

Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.

The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.

If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    var address : bits(32) = R(n) + 4;
    let length : integer = 4*BitCount(registers);
    let within8byte : boolean = AllInAlignedQuantity{32}(address, length, 8);
    for i = 0 to 14 do
        if registers[i] == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS{32}(address, within8byte) = ARBITRARY : bits(32);
            else
                MemS{32}(address, within8byte) = R(i);
            end;
            address = address + 4;
        end;
    end;
    if registers[15] == '1' then
        MemS{32}(address, within8byte) = PCStoreValue();
    end;
    if wback then R(n) = R(n) + 4*BitCount(registers); end;
end;
```

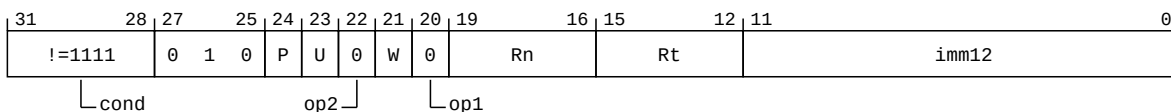
C5.1.224 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

This instruction is used by the alias [PUSH \(single register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), [T3](#), and [T4](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn> {, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STR{<c>}{<q>} <Rt>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STR{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```

if P == '0' && W == '1' then See("STRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

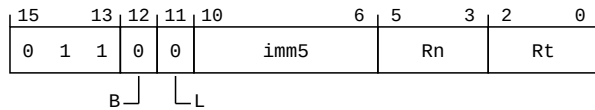
If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1



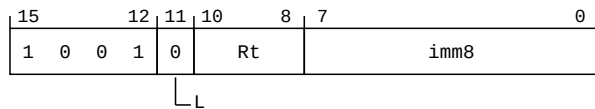
Encoding

STR{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm5::'00')};
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2



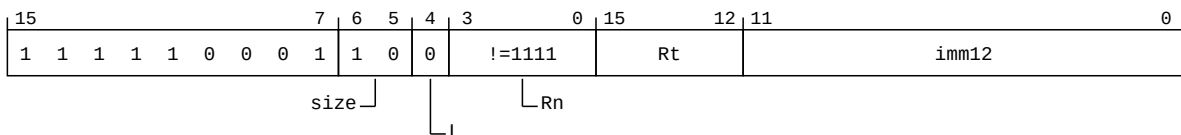
Encoding

STR{<c>}{<q>} <Rt>, [SP{, #<+><imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = 13;
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T3



Encoding

STR{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

STR{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)

Decode for this encoding

```

if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
if t == 15 then UnpredictableProcedure(); end;

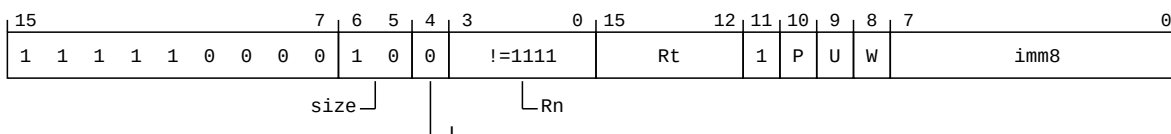
```

CONSTRAINED UNPREDICTABLE behavior

If $t = 15$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is `UNKNOWN`.

T4



Encoding for the Offset variant

Applies when (P == 1 && U == 0 && W == 0)

STR{<c>}{<q>}<Rt>,[<Rn>{,<#-<imm>}]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 1)

STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STR{<c>}{<q>} <Rt>, [<Rn>, # {+/-}<imm>]!

Decode for all variants of this encoding

```

if P == '1' && U == '1' && W == '0' then See("STRT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;

```


CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is `UNKNOWN`.

If `t == 15`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is `UNKNOWN`.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

`<c>`

See [Standard assembler syntax fields](#).

`<q>`

See [Standard assembler syntax fields](#).

`<Rt>`

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “T1”, “T2”, “T3”, “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

`<Rn>`

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1”, “T3”, “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field.

`+/-`

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

`<imm>`

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the “imm5” field as `<imm>/4`.

For the “T2” variant: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the “imm8” field as <imm>/4.

For the “T3” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T4 Offset”, “T4 Post-indexed”, and “T4 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (single register)	A1 Pre-indexed	P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '000000000100'
PUSH (single register)	T4 Pre-indexed	Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100'

Operation

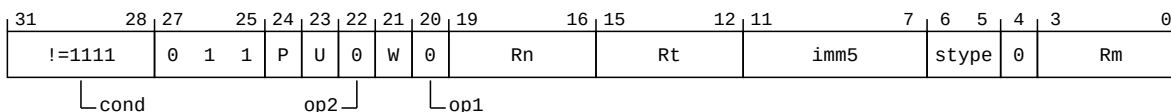
```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{32}(address) = if t == 15 then PCStoreValue() else R(t);
    if wback then R(n) = offset_addr; end;
  end;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{32}(address) = R(t);
    if wback then R(n) = offset_addr; end;
  end;
end;
```

C5.1.225 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
if m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

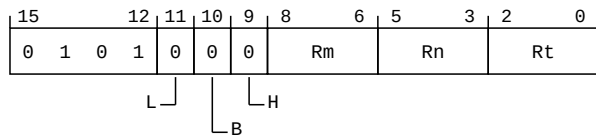
If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If wback && n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1



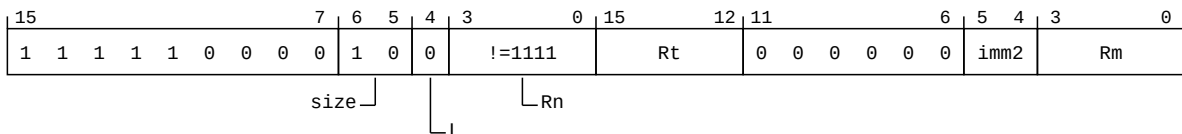
Encoding

STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to `R15` is `UNKNOWN`.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

`<c>`

See [Standard assembler syntax fields](#).

`<q>`

See [Standard assembler syntax fields](#).

`<Rt>`

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “T1” and “T2” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

`<Rn>`

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

`+/-`

Specifies the index register is added to or subtracted from the base register, defaulting to `+` if omitted and encoded in “U”:

U	+/-
0	-
1	+

`<Rm>`

Is the general-purpose index register, encoded in the “Rm” field.

`<shift>`

The shift to apply to the value read from `<Rm>`. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).

`+`

Specifies the index register is added to the base register.

`<imm>`

If present, the size of the left shift to apply to the value from `<Rm>`, in the range 1-3. `<imm>` is encoded in `imm2`. If absent, no shift is specified and `imm2` is encoded as `0b00`.

Operation

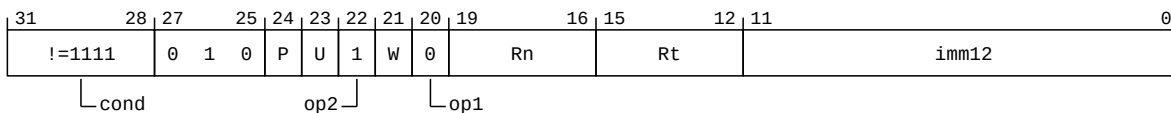
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    var data : bits(32);
    if t == 15 then // Only possible for encoding A1
        data = PCStoreValue();
    else
        data = R(t);
    end;
    MemU{32}(address) = data;
    if wback then R(n) = offset_addr; end;
end;
```

C5.1.226 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, # {+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn>], # {+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, # {+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

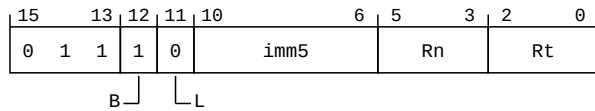
If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1



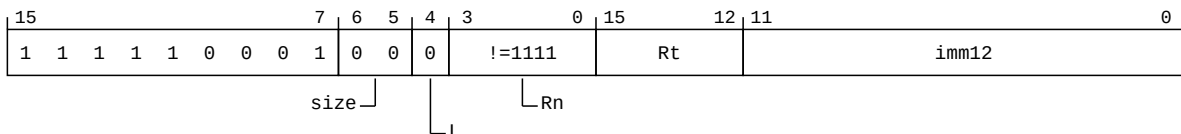
Encoding

STRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm5);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
```

T2



Encoding

STRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

STRB{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

Decode for this encoding

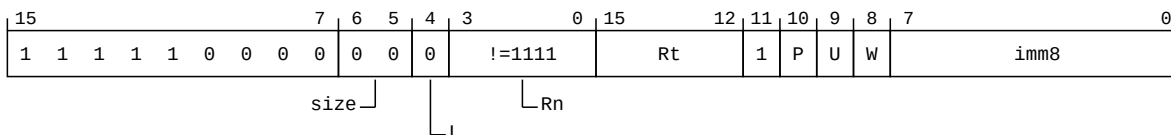
```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```


CONSTRAINED UNPREDICTABLE behavior

If $t == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3



Encoding for the Offset variant

Applies when $(P == 1 \ \&\& \ U == 0 \ \&\& \ W == 0)$

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Encoding for the Post-indexed variant

Applies when $(P == 0 \ \&\& \ W == 1)$

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when $(P == 1 \ \&\& \ W == 1)$

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then See("STRBT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $t == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If $wback \ \&\& \ n == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1”, “T2”, “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.

For the “T2” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
let address : bits(32) = if index then offset_addr else R(n);
```

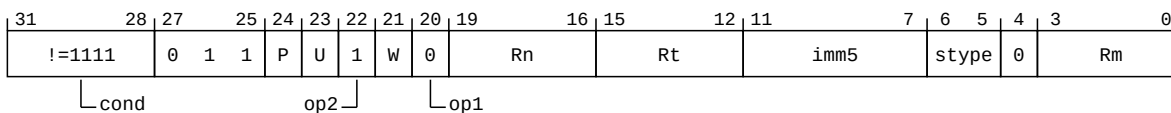
```
MemU{8}(address) = R(t)[7:0];  
if wback then R(n) = offset_addr; end;  
end;
```

C5.1.227 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}]
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRBT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let (shift_t, shift_n) : (SRTtype, integer) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback && n == t, then one of the following behaviors must occur:

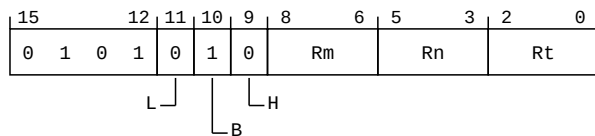
- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1



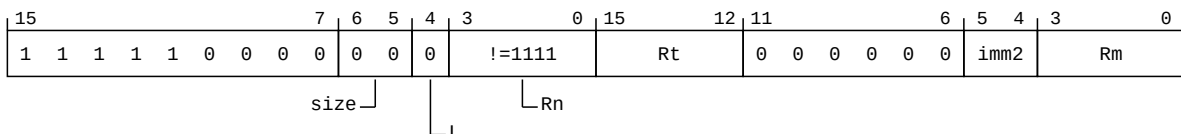
Encoding

STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding

STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{8}(address) = R(t)[7:0];
    if wback then R(n) = offset_addr; end;
end;
```

C5.1.228 STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

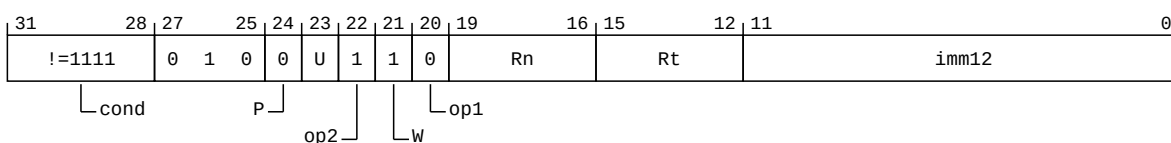
STRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1



Encoding

STRBT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $t == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

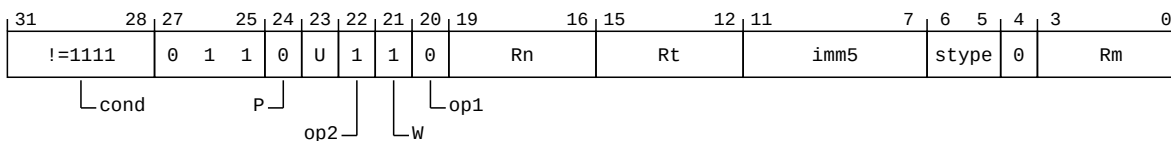
If $n == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2



Encoding

STRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || m == t || m == 15 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

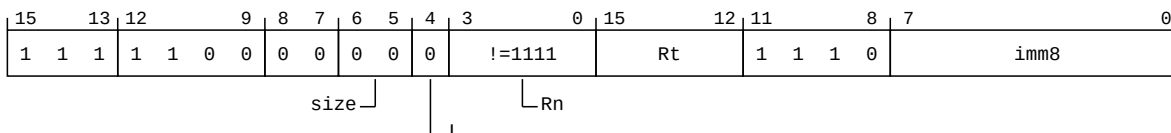
If n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If n == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1



Encoding

STRBT{<c>}{<q>}<Rt>,[<Rn>{,##+<imm>}]

Decode for this encoding

```

if Rn == 'l1ll' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{(imm8)};
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTType = ARBITRARY : SRTType;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $t = 15$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to `R15` is `UNKNOWN`.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

 $\langle R_t \rangle$

For the “A1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “A2” and “T1” variants: is the general-purpose register to be transferred, encoded in the “Rt” field.

 $\langle R_n \rangle$

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end; // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                           else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    MemU_unpriv{8}(address) = R(t)[7:0];
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

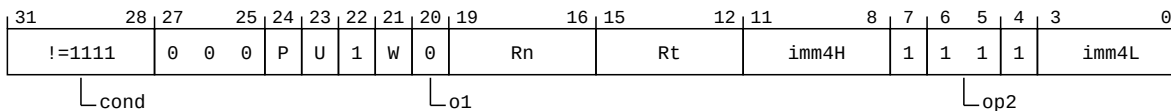
- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction executes as `STRB (immediate)`.

C5.1.229 STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #<+/-><imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #<+/-><imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #<+/-><imm>]!
```

Decode for all variants of this encoding

```
if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm4H::imm4L)};
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;
if t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If t == 15 || t2 == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback && (n == t || n == t2), then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

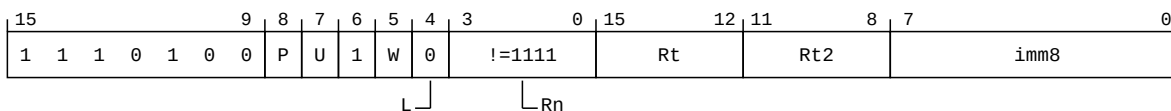
If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `t<0> = '0'`.
- The instruction executes with the additional decode: `t2 = t`.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when `Rt == '1111'`.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

T1



Encoding for the Offset variant

Applies when (`P == 1 && W == 0`)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #<+/-><imm>}]

Encoding for the Post-indexed variant

Applies when (`P == 0 && W == 1`)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #<+/-><imm>

Encoding for the Pre-indexed variant

Applies when (`P == 1 && W == 1`)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #<+/-><imm>]!

Decode for all variants of this encoding

```
if P == '0' && W == '0' then See("Related encodings"); end;
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
if wback && (n == t || n == t2) then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || t == 15 || t2 == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: [Load/store dual](#), [load/store exclusive](#), [table branch](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the second general-purpose register to be transferred. This register must be <R(t+1)>.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

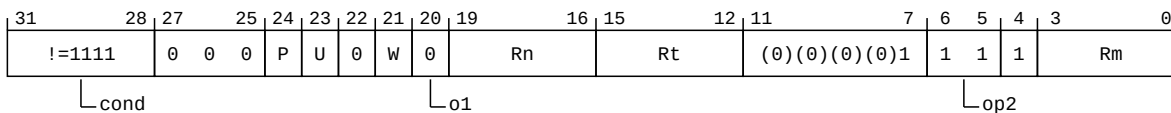
For the “T1 Offset”, “T1 Post-indexed”, and “T1 Pre-indexed” variants: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the “imm8” field as <imm>/4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        var data : bits(64);
        if BigEndian(AccessType_GPR) then
            data[63:32] = R(t);
            data[31:0] = R(t2);
        else
            data[31:0] = R(t);
            data[63:32] = R(t2);
        end;
        MemA{64}(address) = data;
    else
        MemA{32}(address) = R(t);
        MemA{32}(address+4) = R(t2);
    end;
    if wback then R(n) = offset_addr; end;
end;
```


C5.1.230 STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

$$\text{STRD}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle \text{Rt} \rangle, \quad \langle \text{Rt2} \rangle, \quad [\langle \text{Rn} \rangle], \quad \{+/-\} \langle \text{Rm} \rangle$$

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

$$\text{STRD}\{\langle c \rangle\}\{\langle q \rangle\} \quad \langle \text{Rt} \rangle, \langle \text{Rt2} \rangle, [\langle \text{Rn} \rangle, \{+/-\}\langle \text{Rm} \rangle]!$$

Decode for all variants of this encoding

```

if Rt[0] == '1' then UnpredictableProcedure(); end;
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if P == '0' && W == '1' then UnpredictableProcedure(); end;
if t2 == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t || n == t2) then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $t_1 == 15 \parallel t_2 == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If $Rt < 0 > == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $t < 0 > = '0'$.
- The instruction executes with the additional decode: $t2 = t$.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when $Rt == '1111'$.

If $P == '0'$ & $W == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $P = '1'; W = '0'$.
- The instruction executes with the additional decode: $P = '1'; W = '1'$.
- The instruction executes with the additional decode: $P = '0'; W = '0'$.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the first general-purpose register to be transferred, encoded in the “Rt” field. This register must be even-numbered and not R14.

<Rt2>

Is the second general-purpose register to be transferred. This register must be $<R(t+1)>$.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + R(m)) else (R(n) - R(m));
    let address : bits(32) = if index then offset_addr else R(n);
    if IsAlignedSize(address, 8) then
        var data : bits(64);
        if BigEndian(AccessType_GPR) then
            data[63:32] = R(t);
            data[31:0] = R(t2);
        else
            data[31:0] = R(t);
            data[63:32] = R(t2);
        end;
        MemA{64}(address) = data;
    else
        MemA{32}(address) = R(t);
        MemA{32}(address+4) = R(t2);
    end;
    if wback then R(n) = offset_addr; end;
end;
```

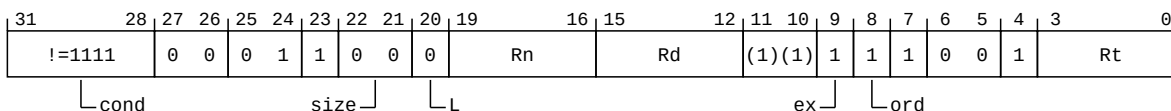
C5.1.231 STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, stores a word from a register to the calculated address if the PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, {#}<imm>}]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = Zeros{}; // Zero offset
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

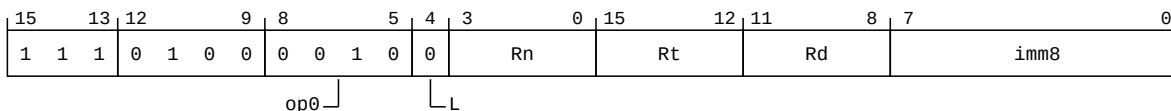
If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1



Encoding

STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, {#}<imm>}]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<imm>

For the “A1” variant: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted.

For the “T1” variant: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n) + imm32;
    if AArch32_ExclusiveMonitorsPass(address,4) then
        MemA{32}(address) = R(t);
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
```

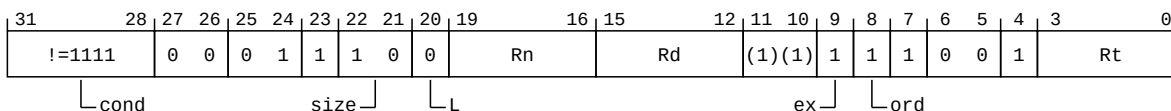
C5.1.232 STREXB

Store Register Exclusive Byte derives an address from a base register value, stores a byte from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

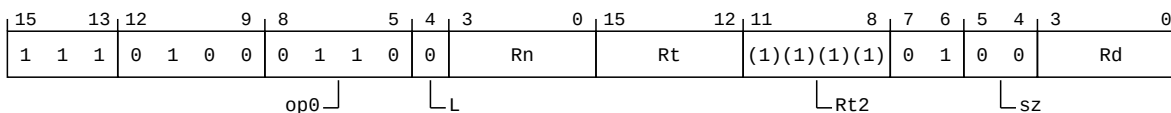
If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1



Encoding

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);  
let t : integer = UInt(Rt);  
let n : integer = UInt(Rn);  
// Armv8-A removes UNPREDICTABLE for R13  
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;  
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If `AArch32.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
let address : bits(32) = R(n);
if AArch32_ExclusiveMonitorsPass(address,1) then
    MemA{8}(address) = R(t)[7:0];
    R(d) = ZeroExtend(32)('0');
else
    R(d) = ZeroExtend(32)('1');
end;
end;
```

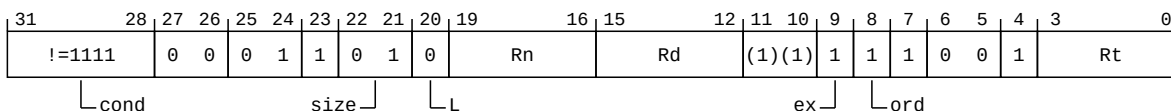
C5.1.233 STREXD

Store Register Exclusive Doubleword derives an address from a base register value, stores a 64-bit doubleword from two registers to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let t : integer = UInt(Rt);
let t2 : integer = t + 1;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
if d == 15 || Rt[0] == '1' || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

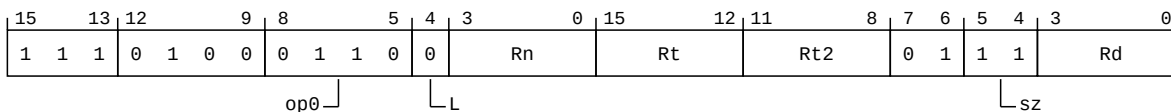
If $Rt<0> == '1'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $Rt<0> = '0'$.
- The instruction executes with the additional decode: $t2 = t$.
- The instruction executes as described, with no change to its behavior and no additional side effects.

If $Rt == '1110'$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction is handled as described in Using R15.

T1



Encoding

STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t || d == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rd> must not be the same as <Rn>, <Rt>, or <Rt2>.

<Rt>

For the “A1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field. <Rt> must be even-numbered and not R14.

For the “T1” variant: is the first general-purpose register to be transferred, encoded in the “Rt” field.

<Rt2>

For the “A1” variant: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.

For the “T1” variant: is the second general-purpose register to be transferred, encoded in the “Rt2” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non doubleword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let address : bits(32) = R(n);

    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    let value : bits(64) = if BigEndian(AccessType_GPR) then R(t)::R(t2) else R(t2)::R(t);

    if AArch32_ExclusiveMonitorsPass(address,8) then
        MemA{64}(address) = value;
        R(d) = ZeroExtend{32}('0');
    else
        R(d) = ZeroExtend{32}('1');
    end;
end;
end;
```

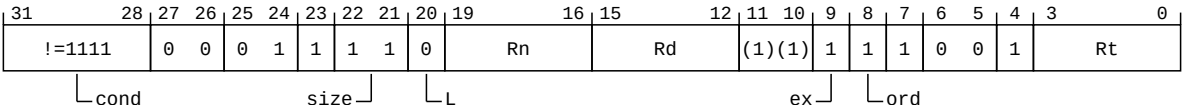
C5.1.234 STREXH

Store Register Exclusive Halfword derives an address from a base register value, stores a halfword from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see [Synchronization and semaphores](#). For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

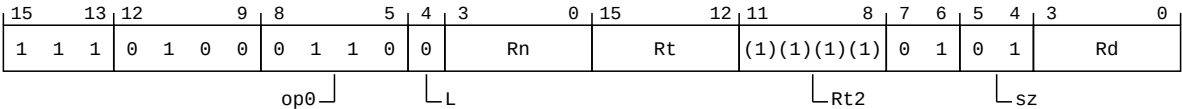
If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1



Encoding

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
let d : integer = UInt(Rd);
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UnpredictableProcedure(); end;
if d == n || d == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $d == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $d == n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the “Rd” field. The value returned is:

- 0 If the operation updates memory.
- 1 If the operation fails to update memory.

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

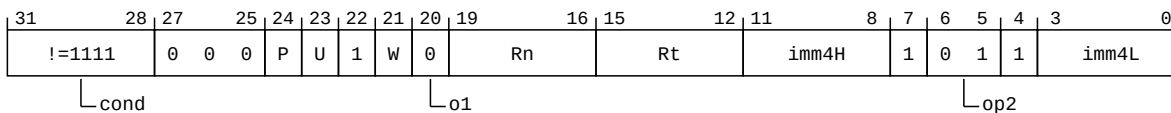
```
if ConditionPassed() then
    EncodingSpecificOperations();
let address : bits(32) = R(n);
if AArch32_ExclusiveMonitorsPass(address,2) then
    MemA{16}(address) = R(t)[15:0];
    R(d) = ZeroExtend{32}('0');
else
    R(d) = ZeroExtend{32}('1');
end;
end;
```

C5.1.235 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

Decode for all variants of this encoding

```
if P == '0' && W == '1' then See("STRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
if t == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

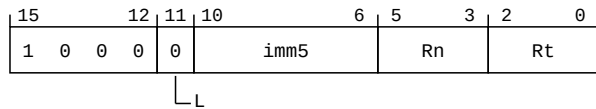
If wback && n == t, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOF`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1



Encoding

STRH{<c>}{<q>}<Rt>,[<Rn>{,<#>{<+><imm>}}]

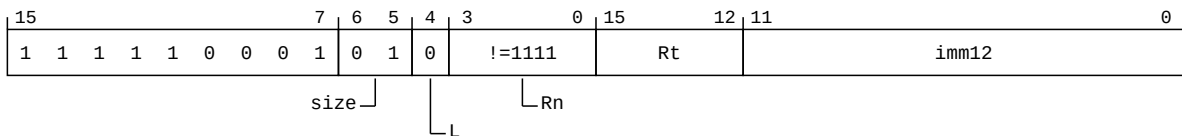
Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm5::'0'};
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;

```

T2



Encoding

STRH{<c>}{<q>}<Rt>,[<Rn>{,<#>{<+><imm>}]

```
STRH{<c>}.W <Rt>, [<Rn> {, #{+<imm>}}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

Decode for this encoding

```

if Rn == 'l1l1' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm12);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;

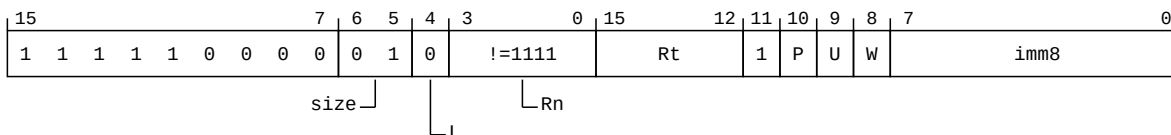
```

CONSTRAINED UNPREDICTABLE behavior

If $t == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3



Encoding for the Offset variant

Applies when $(P == 1 \ \&\& \ U == 0 \ \&\& \ W == 0)$

STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

Encoding for the Post-indexed variant

Applies when $(P == 0 \ \&\& \ W == 1)$

STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

Encoding for the Pre-indexed variant

Applies when $(P == 1 \ \&\& \ W == 1)$

STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

Decode for all variants of this encoding

```

if P == '1' && U == '1' && W == '0' then See("STRHT"); end;
if Rn == '1111' || (P == '0' && W == '0') then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || (wback && n == t) then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $t == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If $wback \ \&\& \ n == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “A1 Offset” variant: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “A1 Post-indexed”, “A1 Pre-indexed”, “T1”, “T2”, “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1 Offset”, “A1 Post-indexed”, and “A1 Pre-indexed” variants: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the “imm5” field as <imm>/2.

For the “T2” variant: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

For the “T3 Offset”, “T3 Post-indexed”, and “T3 Pre-indexed” variants: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm8” field.

+

Specifies the offset is added to the base register.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
```

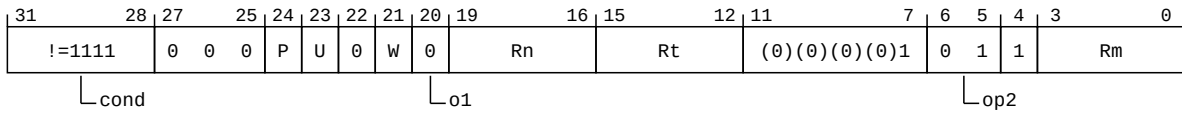
```
        let address : bits(32) = if index then offset_addr else R(n);
        MemU{16}(address) = R(t)[15:0];
        if wback then R(n) = offset_addr; end;
    end;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        let offset_addr : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
        let address : bits(32) = if index then offset_addr else R(n);
        MemU{16}(address) = R(t)[15:0];
        if wback then R(n) = offset_addr; end;
    end;
end;
```

C5.1.236 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (P == 1 && W == 0)

STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]

Encoding for the Post-indexed variant

Applies when (P == 0 && W == 0)

STRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Encoding for the Pre-indexed variant

Applies when (P == 1 && W == 1)

STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!

Decode for all variants of this encoding

```

if P == '0' && W == '1' then See("STRHT"); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = (P == '1');
let add : boolean = (U == '1');
let wback : boolean = (P == '0') || (W == '1');
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
if t == 15 || m == 15 then UnpredictableProcedure(); end;
if wback && (n == 15 || n == t) then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If t == 15, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If wback && n == t, then one of the following behaviors must occur:

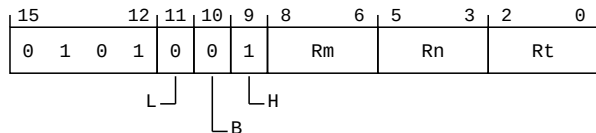
- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1



Encoding

STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

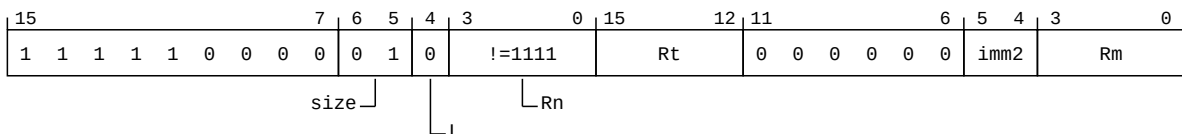
Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;

```

T2



Encoding

STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let index : boolean = TRUE;
let add : boolean = TRUE;
let wback : boolean = FALSE;
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = UInt(imm2);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

For the “Offset”, “Post-indexed”, and “Pre-indexed” variants: is the general-purpose base register, encoded in the “Rn” field. The PC can be used in the offset variant, but this is deprecated.

For the “T1” and “T2” variants: is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	–
1	+

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the index register is added to the base register.

<imm>

If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let offset : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if index then offset_addr else R(n);
    MemU{16}(address) = R(t)[15:0];
    if wback then R(n) = offset_addr; end;
end;
```

C5.1.237 STRHT

Store Register Halfword Unprivileged stores a halfword from a register to memory. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

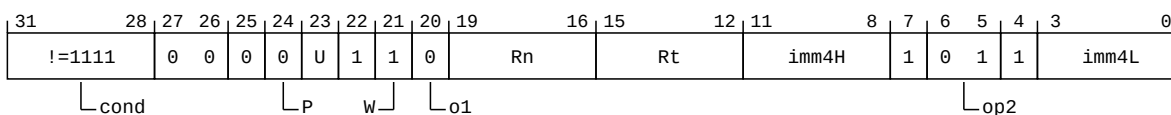
STRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1



Encoding

```
STRHT{<c>}{<q>} <Rt>, [<Rn>] {, #(+/-)<imm>}
```

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm4H::imm4L);
let m : integer = ARBITRARY : integer;
if t == 15 || n == 15 || n == t then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $t == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If $n == t$, then one of the following behaviors must occur:

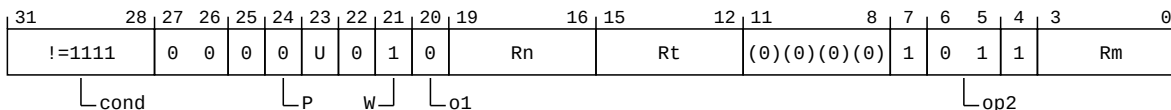
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2



Encoding

STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Decode for this encoding

```
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let imm32 : bits(32) = ARBITRARY : bits(32);
if t == 15 || n == 15 || m == t || m == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $t == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

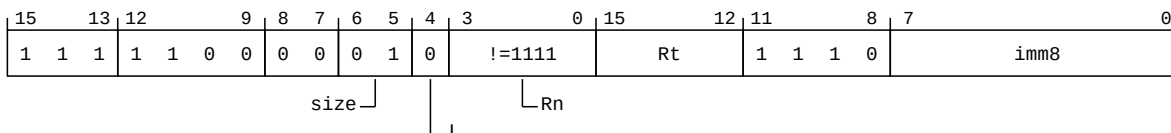
If $n == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1



Encoding

STRHT{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

Decode for this encoding

```

if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $t == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the “imm4H:imm4L” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end; // Hyp mode
    let offset : bits(32) = if register_form then R(m) else imm32;
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    MemU_unpriv{16}(address) = R(t)[15:0];
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as STRH (immediate).

C5.1.238 STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

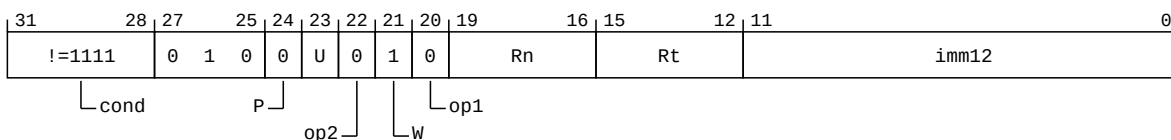
STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1



Encoding

STRT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm12);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
if n == 15 || n == t then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

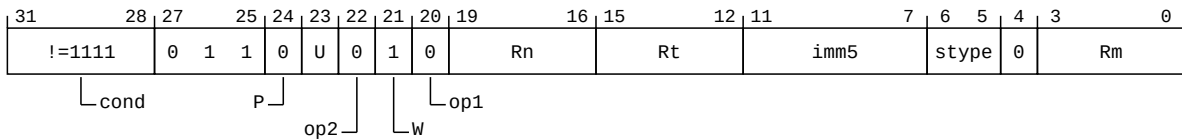
If $n == t$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If $n == 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if $\text{bit}[24] == 1$ and $\text{bit}[21] == 0$. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2



Encoding

STRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

Decode for this encoding

```

let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let postindex : boolean = TRUE;
let add : boolean = (U == '1');
let register_form : boolean = TRUE;
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
let imm32 : bits(32) = ARBITRARY : bits(32);
if n == 15 || n == t || m == 15 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

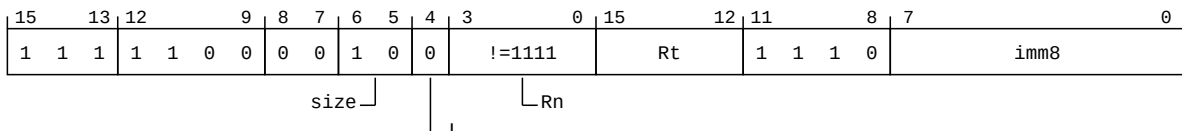
If $n == t$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The store instruction executes but the value stored is **UNKNOWN**.

If $n = 15$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in Using R15.
- The instruction is treated as if $\text{bit}[24] = 1$ and $\text{bit}[21] = 0$. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1



Encoding

STRT{<c>} {<q>} <Rt>, [<Rn> {, # {+<imm>}]

Decode for this encoding

```
if Rn == '1111' then Undefined(); end;
let t : integer = UInt(Rt);
let n : integer = UInt(Rn);
let postindex : boolean = FALSE;
let add : boolean = TRUE;
let register_form : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm8);
let m : integer = ARBITRARY : integer;
let shift_n : integer = ARBITRARY : integer;
let shift_t : SRTYPE = ARBITRARY : SRTYPE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

For the “A1” and “A2” variants: is the general-purpose register to be transferred, encoded in the “Rt” field. The PC can be used, but this is deprecated.

For the “T1” variant: is the general-purpose register to be transferred, encoded in the “Rt” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

For the “A1” variant: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For the “A2” variant: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

For the “A1” variant: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the “imm12” field.

For the “T1” variant: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the “imm8” field.

<Rm>

Is the general-purpose index register, encoded in the “Rm” field.

<shift>

The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see [Shifts applied to a register](#).

+

Specifies the offset is added to the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UnpredictableProcedure(); end;           // Hyp mode
    let offset : bits(32) = (if register_form then Shift{32}(R(m), shift_t, shift_n, PSTATE.C)
                             else imm32);
    let offset_addr : bits(32) = if add then (R(n) + offset) else (R(n) - offset);
    let address : bits(32) = if postindex then R(n) else offset_addr;
    var data : bits(32);
    if t == 15 then // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R(t);
    end;
    MemU_unpriv{32}(address) = data;
    if postindex then R(n) = offset_addr; end;
end;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.EL == EL2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as STR (immediate).

C5.1.239 SUB (immediate, from PC)

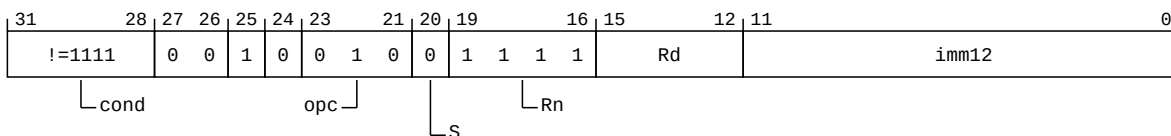
Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This is an alias of [ADR](#). This means:

- The encodings in this description are named to match the encodings of [ADR](#).
- The description of [ADR](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A2](#)) and T32 ([T2](#)).

A2



Encoding for the A2 variant

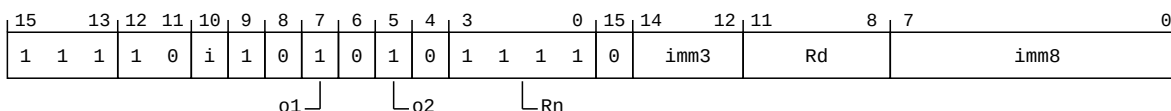
SUB{<c>}{<q>} <Rd>, PC, #<const>

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

and is the preferred disassembly when imm12 == '000000000000'.

T2



Encoding for the T2 variant

SUB{<c>}{<q>} <Rd>, PC, #<imm12>

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

and is the preferred disassembly when i :: imm3 :: imm8 == '000000000000'.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A2” variant: is the general-purpose destination register, encoded in the “Rd” field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For the “T2” variant: is the general-purpose destination register, encoded in the “Rd” field.

<const>

An immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

Operation

The description of [ADR](#) gives the operational pseudocode for this instruction.

C5.1.240 SUB, SUBS (immediate)

Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register.

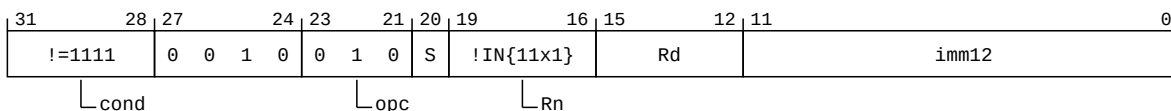
If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SUBS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores `PSTATE` from `SPSR_<current_mode>`.
 - The PE checks `SPSR_<current_mode>` for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode, except for encoding T5 with <imm8> set to zero, which is the encoding for the `ERET` instruction, see `ERET`.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), [T3](#), [T4](#), and [T5](#)).

A1



Encoding for the SUB variant

Applies when (S == 0 && Rn != 11x1)

SUB{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Encoding for the SUBS variant

Applies when (S == 1 && Rn != 1101)

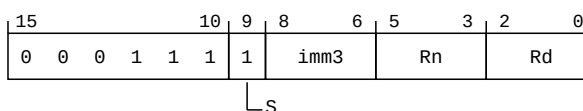
SUBS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

Decode for all variants of this encoding

```

if Rn == '1111' && S == '0' then See("ADR"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
    
```

T1



Encoding

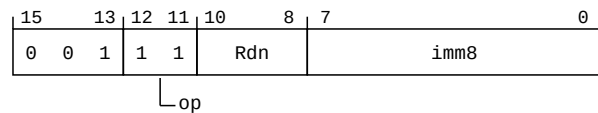
SUB<c>{<q>} <Rd>, <Rn>, #<imm3> // (InITBlock())

SUBS{<q>} <Rd>, <Rn>, #<imm3> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm3);
```

T2



Encoding

SUB<c>{<q>} <Rdn>, #<imm8> // (InITBlock() && UInt(Rdn) < 8 && UInt(imm8) < 8)

SUB<c>{<q>} {<Rdn>, }<Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)

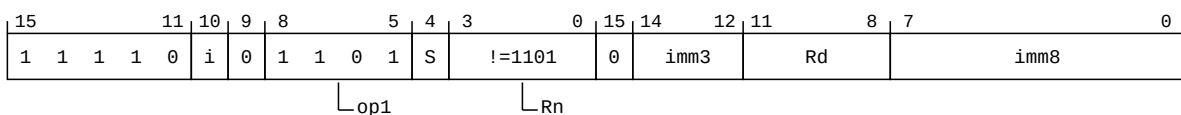
SUBS{<q>} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)

SUBS{<q>} {<Rdn>, }<Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rdn);
let n : integer = UInt(Rdn);
let setflags : boolean = !InITBlock();
let imm32 : bits(32) = ZeroExtend{}(imm8);
```

T3



Encoding for the SUB variant

Applies when (S == 0)

SUB{<c>}{<q>} {<Rd>, }<Rn>, #<const>

SUB<c>.W {<Rd>, }<Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1
→ or T2)

Encoding for the SUBS variant

Applies when (S == 1 && Rd != 1111)

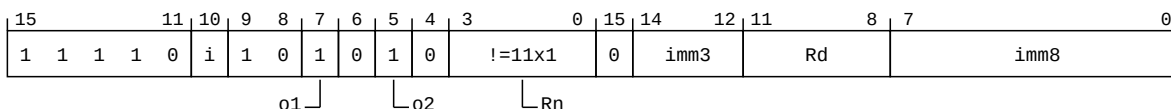
SUBS{<c>}{<q>} {<Rd>, }<Rn>, #<const>

SUBS.W {<Rd>, }<Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
↪T2)

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (immediate)"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UnpredictableProcedure(); end;
```

T4



Encoding

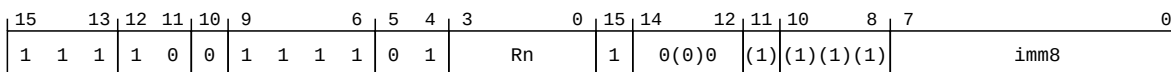
SUB{<c>}{<q>} {<Rd>, }<Rn>, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

SUBW{<c>}{<q>} {<Rd>, }<Rn>, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

Decode for this encoding

```
if Rn == '1111' then See("ADR"); end;
if Rn == '1101' then See("SUB (SP minus immediate)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UnpredictableProcedure(); end;
```

T5



Encoding

Applies when (!(Rn == 1110 && imm8 == 00000000))

SUBS{<c>}{<q>} PC, LR, #<imm8>

Decode for this encoding

```
if Rn == '1110' && IsZero(imm8) then See("ERET"); end;
let d : integer = 15;
let n : integer = UInt(Rn);
let setflags : boolean = TRUE;
let imm32 : bits(32) = ZeroExtend{(imm8)};
if n != 14 then UnpredictableProcedure(); end;
if InITBlock() && !LastInITBlock() then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [SUBS PC, LR and related instructions \(A32\)](#) and [SUBS PC, LR and related instructions \(T32\)](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 SUB” and “A1 SUBS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SUBS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`. Arm deprecates use of this instruction unless <Rn> is the LR.

For the “T1”, “T3 SUB”, “T3 SUBS”, and “T4” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 SUB”, “A1 SUBS”, and “T4” variants: is the general-purpose source register, encoded in the “Rn” field. If the SP is used, see [SUB \(SP minus immediate\)](#). If the PC is used, see [ADR](#).

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

For the “T3 SUB” and “T3 SUBS” variants: is the general-purpose source register, encoded in the “Rn” field. If the SP is used, see [SUB \(SP minus immediate\)](#).

<const>

For the “A1 SUB” and “A1 SUBS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T3 SUB” and “T3 SUBS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

<imm3>

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “imm3” field.

<Rdn>

Is the general-purpose source and destination register, encoded in the “Rdn” field.

<imm8>

For the “T2” variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field.

For the “T5” variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field. If <Rn> is the LR, and zero is used, see ERET.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

In the T32 instruction set, `MOVS{<c>}{<q>} PC, LR` is a pseudo-instruction for `SUBS{<c>}{<q>} PC, LR, #0`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(imm32), '1');
    if d == 15 then
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

C5.1.241 SUB, SUBS (register)

Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

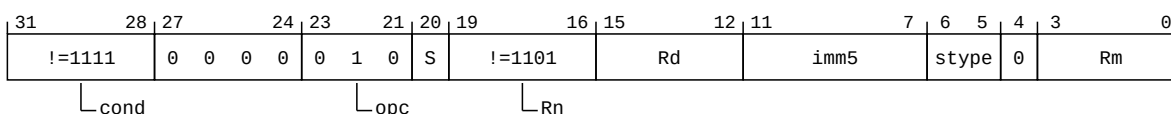
If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the SUB, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

SUB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SUB, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

SUB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

Encoding for the SUBS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

SUBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SUBS, shift or rotate by value variant

Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

SUBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

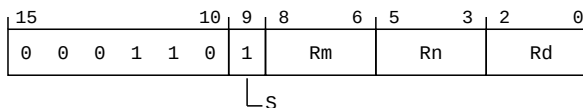
Decode for all variants of this encoding

```

if Rn == '1101' then See("SUB (SP minus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);

```

T1



Encoding

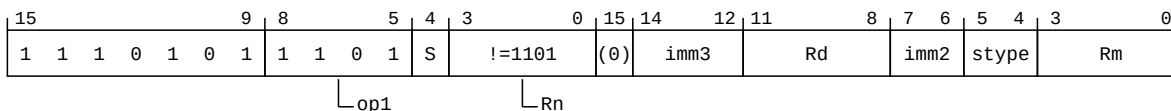
SUB<c>{<q>} <Rd>, <Rn>, <Rm> // (InITBlock())

SUBS{<q>} {<Rd>, }<Rn>, <Rm> // (Outside IT block)

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = !InITBlock();
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the SUB, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

SUB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SUB, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

SUB<c>.W {<Rd>, }<Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Encoding for the SUBS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

SUBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, RRX

Encoding for the SUBS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, <shift> #<amount>}

SUBS.W {<Rd>, }<Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```

if Rd == '1111' && S == '1' then See("CMP (register)"); end;
if Rn == '1101' then See("SUB (SP minus register)"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SUBS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_current_mode`.

For the “T1”, “T2 SUB, rotate right with extend”, “T2 SUB, shift or rotate by value”, “T2 SUBS, rotate right with extend”, and “T2 SUBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the same as <Rn>.

<Rn>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated. If the SP is used, see [SUB \(SP minus register\)](#).

For the “T1” variant: is the first general-purpose source register, encoded in the “Rn” field.

For the “T2 SUB, rotate right with extend”, “T2 SUB, shift or rotate by value”, “T2 SUBS, rotate right with extend”, and “T2 SUBS, shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. If the SP is used, see [SUB \(SP minus register\)](#).

<Rm>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 SUB, rotate right with extend”, “T2 SUB, shift or rotate by value”, “T2 SUBS, rotate right with extend”, and “T2 SUBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 SUB, shift or rotate by value” and “A1 SUBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

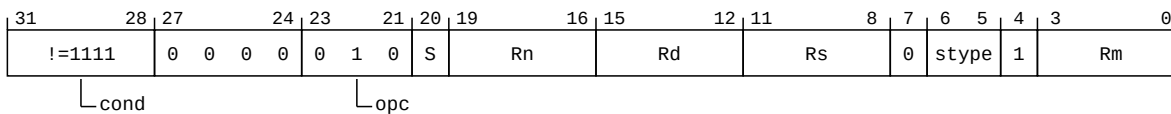
For the “T2 SUB, shift or rotate by value” and “T2 SUBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```

C5.1.242 SUB, SUBS (register-shifted register)

Subtract (register-shifted register) subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.



Encoding for the Flag setting variant

Applies when (S == 1)

SUBS{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Encoding for the Not flag setting variant

Applies when (S == 0)

SUB{<c>}{<q>} {<Rd>, }<Rn>, <Rm>, <shift> <Rs>

Decode for all variants of this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let setflags : boolean = (S == '1');
let shift_t : SRTYPE = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(n), NOT(shifted), '1');
    R(d) = result;
    if setflags then
        PSTATE.[N,Z,C,V] = nzcvc;
    end;
end;
```

C5.1.243 SUB, SUBS (SP minus immediate)

Subtract from SP (immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

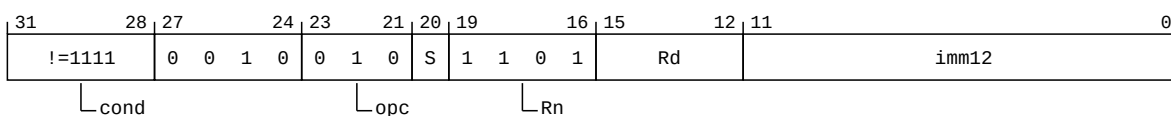
If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the SUB variant

Applies when (S == 0)

SUB{<c>}{<q>} {<Rd>}, }SP, #<const>

Encoding for the SUBS variant

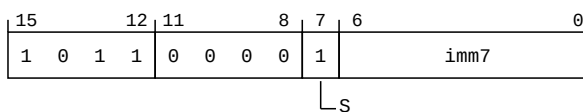
Applies when (S == 1)

SUBS{<c>}{<q>} {<Rd>}, }SP, #<const>

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = A32ExpandImm(imm12);
```

T1



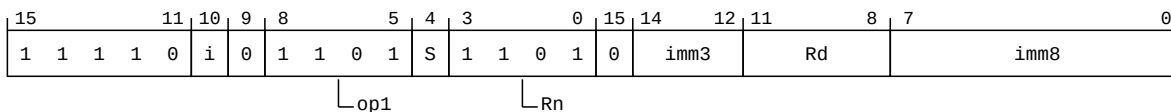
Encoding

SUB{<c>}{<q>} {SP, }SP, #<imm7>

Decode for this encoding

```
let d : integer = 13;
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(imm7::'00');
```

T2



Encoding for the SUB variant

Applies when (S == 0)

```
SUB{<c>}{<q>} {<Rd>, }SP, #<const>
```

```
SUB{<c>}.W {<Rd>, }SP, #<const> // (<Rd>, <const> can be represented in T1)
```

Encoding for the SUBS variant

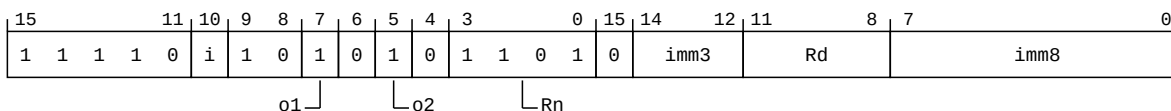
Applies when (S == 1 && Rd != 1111)

```
SUBS{<c>}{<q>} {<Rd>, }SP, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (immediate)"); end;
let d : integer = UInt(Rd);
let setflags : boolean = (S == '1');
let imm32 : bits(32) = T32ExpandImm(i::imm3::imm8);
if d == 15 && !setflags then UnpredictableProcedure(); end;
```

T3



Encoding

```
SUB{<c>}{<q>} {<Rd>, }SP, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)
```

```
SUBW{<c>}{<q>} {<Rd>, }SP, #<imm12> // (<imm12> can be represented in T1, T2, or T3)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let setflags : boolean = FALSE;
let imm32 : bits(32) = ZeroExtend{}(i::imm3::imm8);
if d == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 SUB” and “A1 SUBS” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SUBS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`. Arm deprecates use of this instruction unless <Rn> is the LR.

For the “T2 SUB”, “T2 SUBS”, and “T3” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP.

<const>

For the “A1 SUB” and “A1 SUBS” variants: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T2 SUB” and “T2 SUBS” variants: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

SP

Is the stack pointer.

<imm7>

Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the “imm7” field as <imm7>/4.

<imm12>

Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the “i:imm3:imm8” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), NOT(imm32), '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
end;

```

C5.1.244 SUB, SUBS (SP minus register)

Subtract from SP (register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

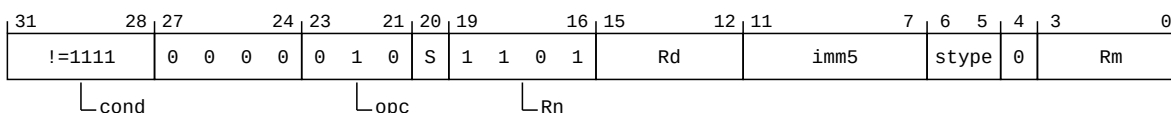
If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
 - The PE branches to the address written to the PC, and restores **PSTATE** from SPSR_<current_mode>.
 - The PE checks SPSR_<current_mode> for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the SUB, rotate right with extend variant

Applies when (S == 0 && imm5 == 00000 && stype == 11)

SUB{<c>}{<q>} {<Rd>, }SP, <Rm> , RRX

Encoding for the SUB, shift or rotate by value variant

Applies when (S == 0 && !(imm5 == 00000 && stype == 11))

SUB{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}

Encoding for the SUBS, rotate right with extend variant

Applies when (S == 1 && imm5 == 00000 && stype == 11)

SUBS{<c>}{<q>} {<Rd>, }SP, <Rm> , RRX

Encoding for the SUBS, shift or rotate by value variant

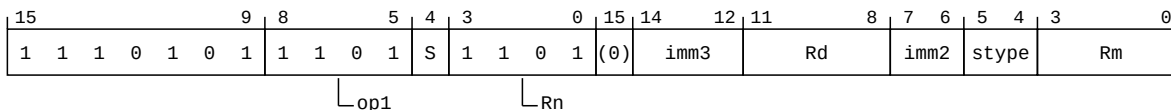
Applies when (S == 1 && !(imm5 == 00000 && stype == 11))

SUBS{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm5);
```

T1



Encoding for the SUB, rotate right with extend variant

Applies when (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

SUB{<c>}{<q>} {<Rd>, }SP, <Rm>, RRX

Encoding for the SUB, shift or rotate by value variant

Applies when (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUB{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>} // (Preferred syntax)

SUB{<c>}.W {<Rd>, }SP, <Rm>

Encoding for the SUBS, rotate right with extend variant

Applies when (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

SUBS{<c>}{<q>} {<Rd>, }SP, <Rm>, RRX

Encoding for the SUBS, shift or rotate by value variant

Applies when (S == 1 && Rd != 1111 && !(imm3 == 000 && imm2 == 00 && stype == 11))

SUBS{<c>}{<q>} {<Rd>, }SP, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then See("CMP (register)"); end;
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
let (shift_t, shift_n) : (SRTyp, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- For the SUBS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.

For the “T1 SUB, rotate right with extend”, “T1 SUB, shift or rotate by value”, “T1 SUBS, rotate right with extend”, and “T1 SUBS, shift or rotate by value” variants: is the general-purpose destination register, encoded in the “Rd” field. If omitted, this register is the SP.

<Rm>

For the “A1 SUB, rotate right with extend”, “A1 SUB, shift or rotate by value”, “A1 SUBS, rotate right with extend”, and “A1 SUBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1 SUB, rotate right with extend”, “T1 SUB, shift or rotate by value”, “T1 SUBS, rotate right with extend”, and “T1 SUBS, shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 SUB, shift or rotate by value” and “A1 SUBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

For the “T1 SUB, shift or rotate by value” and “T1 SUBS, shift or rotate by value” variants: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shifted : bits(32) = Shift{}(R(m), shift_t, shift_n, PSTATE.C);
    let (result, nzcvc) : (bits(32), bits(4)) = AddWithCarry{32}(R(13), NOT(shifted), '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
        end;
    else
        R(d) = result;
        if setflags then
            PSTATE.[N,Z,C,V] = nzcvc;
        end;
    end;
end;
```


Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<imm>

For the “A1” variant: is a 24-bit unsigned immediate, in the range 0 to 16777215, encoded in the “imm24” field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm24 in software, for example to determine the required service.

For the “T1” variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm8 in software, for example to determine the required service.

Operation

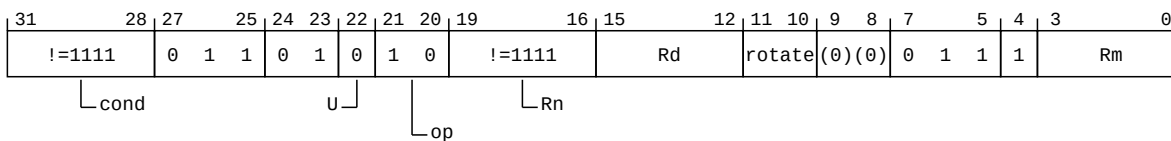
```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32_CallSupervisor(imm32[15:0]);
end;
```

C5.1.246 SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



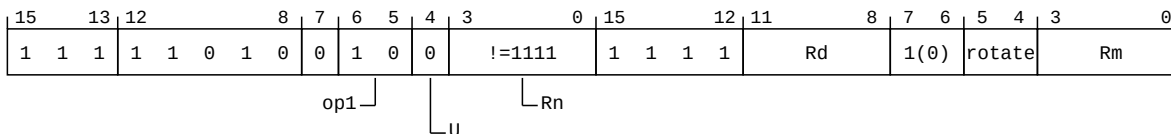
Encoding

SXTAB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SXTAB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + SignExtend{32}(rotated[7:0]);
end;
```

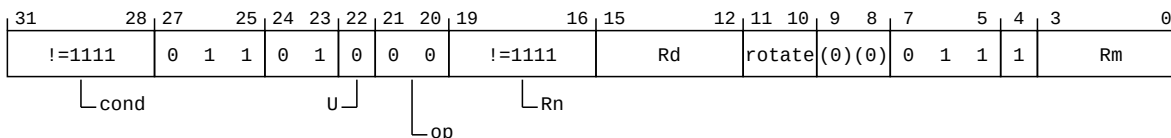
C5.1.247 SXTAB16

Signed Extend and Add Byte 16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



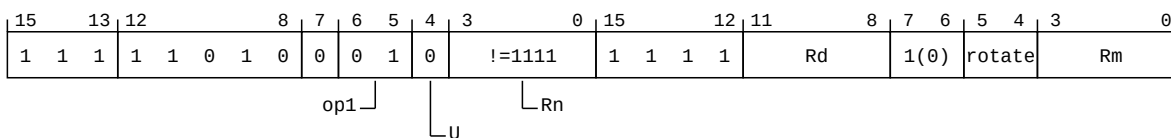
Encoding

SXTAB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SXTAB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

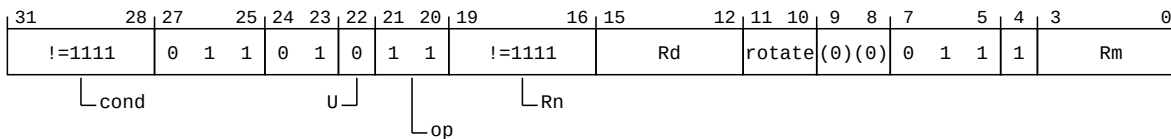
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0] = R(n)[15:0] + SignExtend{16}(rotated[7:0]);
    R(d)[31:16] = R(n)[31:16] + SignExtend{16}(rotated[23:16]);
end;
```

C5.1.248 SXTAH

Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



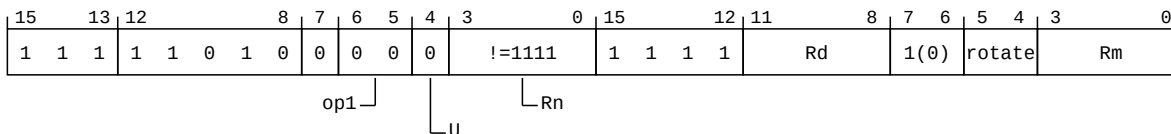
Encoding

SXTAH{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

SXTAH{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("SXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

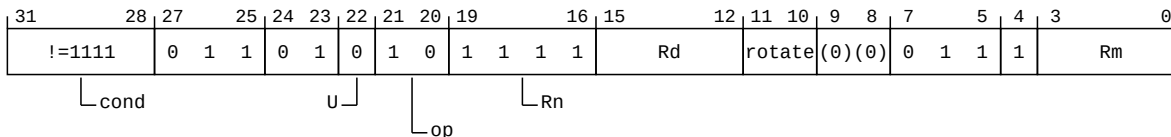
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + SignExtend{32}(rotated[15:0]);
end;
```

C5.1.249 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



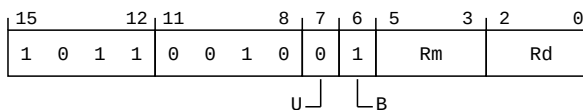
Encoding

SXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



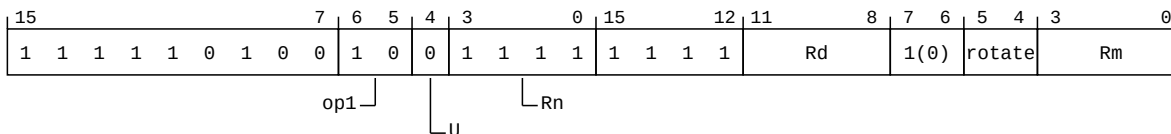
Encoding

SXTB{<c>}{<q>} {<Rd>, }<Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2



Encoding

```
SXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

SXTB{<c>}.W {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>
See [Standard assembler syntax fields](#).

<q>
See [Standard assembler syntax fields](#).

<Rd>
Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>
Is the general-purpose source register, encoded in the “Rm” field.

<amount>
Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

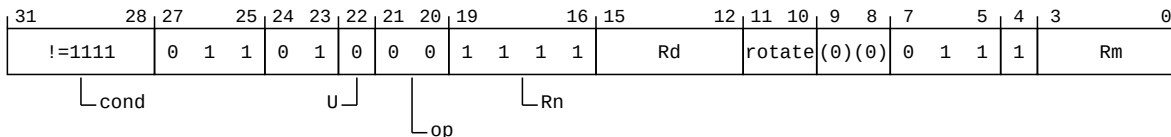
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = SignExtend{32}(rotated[7:0]);
end;
```

C5.1.250 SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



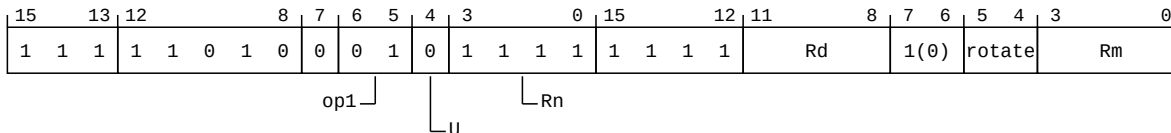
Encoding

```
SXTB16{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
SXTB16{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

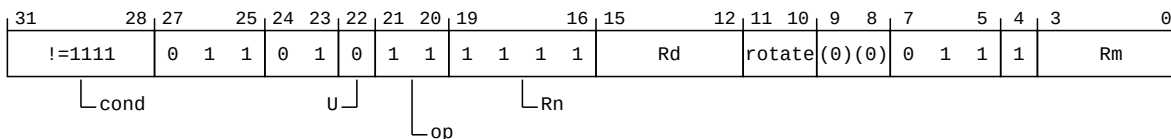
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0] = SignExtend{16}(rotated[7:0]);
    R(d)[31:16] = SignExtend{16}(rotated[23:16]);
end;
```

C5.1.251 SXTB

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



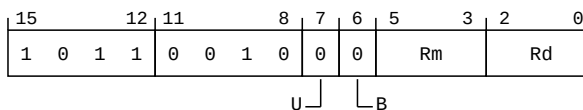
Encoding

SXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



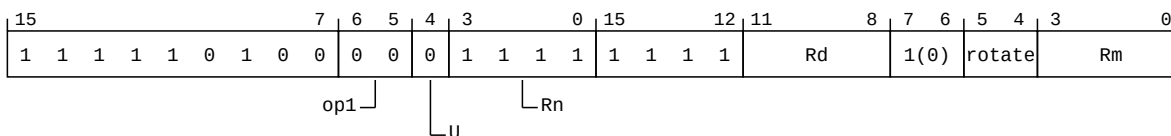
Encoding

SXTB{<c>}{<q>} {<Rd>, }<Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2



Encoding

SXTH{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}

SXTH{<c>}.W {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>
See [Standard assembler syntax fields](#).

<q>
See [Standard assembler syntax fields](#).

<Rd>
Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>
Is the general-purpose source register, encoded in the “Rm” field.

<amount>
Is the rotate amount, encoded in “rotate”:

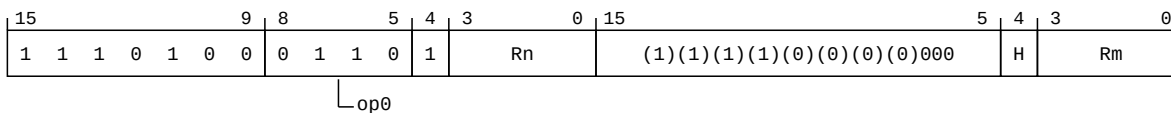
rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = SignExtend{32}(rotated[15:0]);
end;
```

C5.1.252 **TBB, TBH**

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.



Encoding for the Byte variant

Applies when $(H == 0)$

```
TBB{<c>}{<q>}    [<Rn>, <Rm>] // (Outside or last in IT block)
```

Encoding for the Halfword variant

Applies when $(H == 1)$

```
TBH{<c>}{<q>}    [<Rn>, <Rm>, LSL #1] // (Outside or last in IT block)
```

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let is_tbh : boolean = (H == '1');
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UnpredictableProcedure(); end;
if !IntBlock() && !LastIntBlock() then UnpredictableProcedure(); end;
```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

 $\langle R_n \rangle$

Is the general-purpose base register holding the address of the table of branch lengths, encoded in the “Rn” field. The PC can be used. If it is, the table immediately follows this instruction.

<Rm>

For the “Byte” variant: is the general-purpose index register, encoded in the “Rm” field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.

For the “Halfword” variant: is the general-purpose index register, encoded in the “Rm” field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

Operation

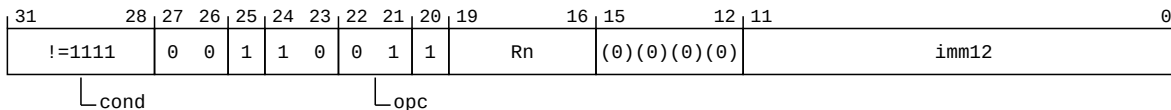
```
if ConditionPassed() then
    EncodingSpecificOperations();
var halfwords : integer;
if is_tbh then
    halfwords = UInt(MemU{16}(R(n)+LSL(R(m), 1)));
else
    halfwords = UInt(MemU{8}(R(n)+R(m)));
end;
BranchWritePC(PC32() + 2*halfwords, BranchType_INDIR);
end;
```

C5.1.253 TEQ (immediate)

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



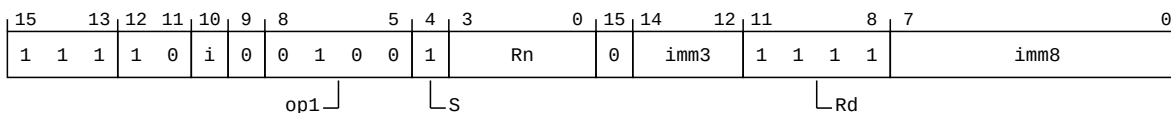
Encoding

TEQ{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding

TEQ{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

For the “A1” variant: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1” variant: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1” variant: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                            else T32ExpandImm_C(imm, PSTATE.C));

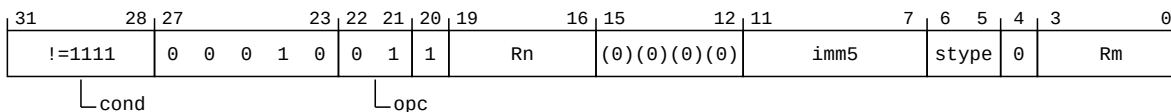
    let result : bits(32) = R(n) XOR imm32;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

C5.1.254 TEQ (register)

Test Equivalence (register) performs a bitwise exclusive-OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

TEQ{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

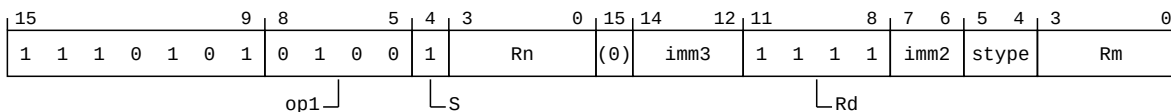
Applies when (!(imm5 == 00000 && stype == 11))

TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1



Encoding for the Rotate right with extend variant

Applies when (imm3 == 000 && imm2 == 00 && stype == 11)

TEQ{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

Applies when (!(imm3 == 000 && imm2 == 00 && stype == 11))

TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1 Rotate right with extend” and “T1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1 Rotate right with extend” and “T1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

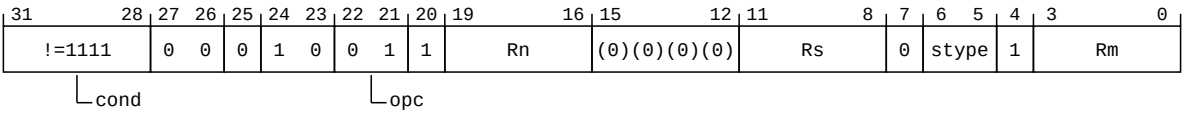
For the “T1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

C5.1.255 **TEQ (register-shifted register)**

Test Equivalence (register-shifted register) performs a bitwise exclusive-OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.



Encoding

TEQ{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SRTYPE = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<type>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

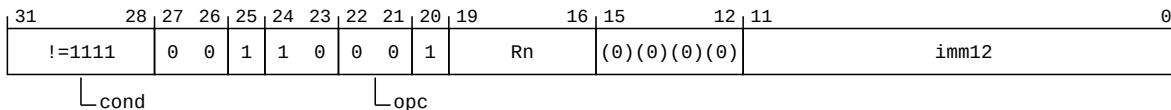
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) XOR shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

C5.1.256 TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



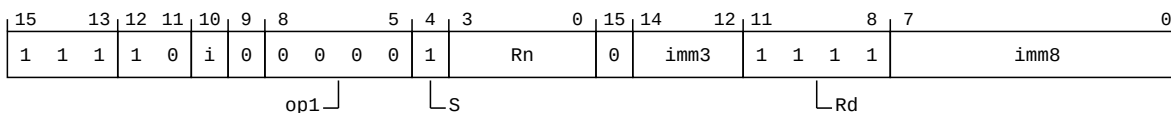
Encoding

TST{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = TRUE;
let imm : bits(12) = imm12;
```

T1



Encoding

TST{<c>}{<q>} <Rn>, #<const>

Decode for this encoding

```
let n : integer = UInt(Rn);
let a32 : boolean = FALSE;
let imm : bits(12) = i::imm3::imm8;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

For the “A1” variant: is the general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1” variant: is the general-purpose source register, encoded in the “Rn” field.

<const>

For the “A1” variant: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For the “T1” variant: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (imm32, carry) : (bits(32), bit) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                                            else T32ExpandImm_C(imm, PSTATE.C));

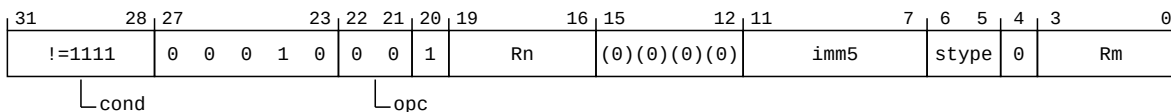
    let result : bits(32) = R(n) AND imm32;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

C5.1.257 TST (register)

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Rotate right with extend variant

Applies when (imm5 == 00000 && stype == 11)

TST{<c>}{<q>} <Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

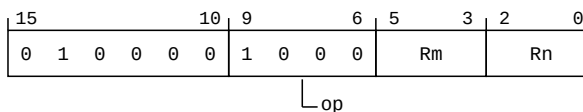
Applies when (!(imm5 == 00000 && stype == 11))

TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

Decode for all variants of this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTYPE, integer) = DecodeImmShift(stype, imm5);
```

T1



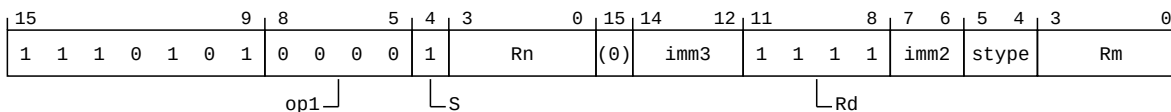
Encoding

TST{<c>}{<q>} <Rn>, <Rm>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let shift_t : SRTYPE = SRTYPE_LSL;
let shift_n : integer = 0;
```

T2



Encoding for the Rotate right with extend variant

Applies when (imm3 == 000 && imm2 == 00 && stype == 11)

TST{<c>}{<q>}<Rn>, <Rm>, RRX

Encoding for the Shift or rotate by value variant

Applies when `(!(imm3 == 000 && imm2 == 00 && stype == 11))`

TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

TST{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

Decode for all variants of this encoding

```

let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(stype, imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

 $\langle R_n \rangle$

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 Rotate right with extend”, and “T2 Shift or rotate by value” variants: is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

For the “A1 Rotate right with extend” and “A1 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field. The PC can be used, but this is deprecated.

For the “T1”, “T2 Rotate right with extend”, and “T2 Shift or rotate by value” variants: is the second general-purpose source register, encoded in the “Rm” field.

<shift>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount>

For the “A1 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the “imm5” field as <amount> modulo 32.

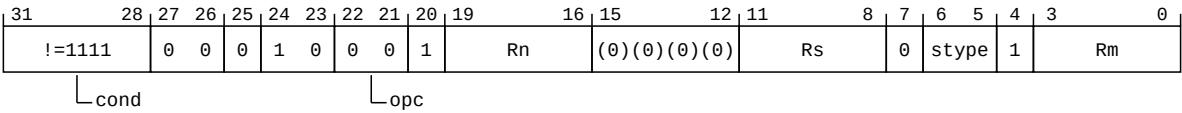
For the “T2 Shift or rotate by value” variant: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the “imm3:imm2” field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

C5.1.258 TST (register-shifted register)

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.



Encoding

TST{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

Decode for this encoding

```
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let s : integer = UInt(Rs);
let shift_t : SType = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<type>

Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs>

Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the “Rs” field.

Operation

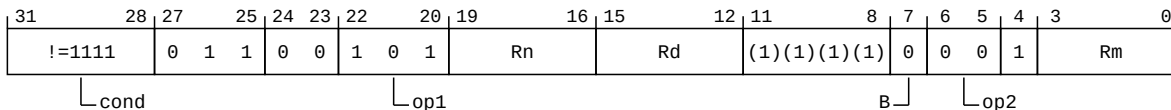
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let shift_n : integer = UInt(R(s)[7:0]);
    let (shifted, carry) : (bits(32), bit) = Shift_C{32}(R(m), shift_t, shift_n, PSTATE.C);
    let result : bits(32) = R(n) AND shifted;
    PSTATE.N = result[31];
    PSTATE.Z = IsZeroBit{32}(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
end;
```

C5.1.259 UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results of the additions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



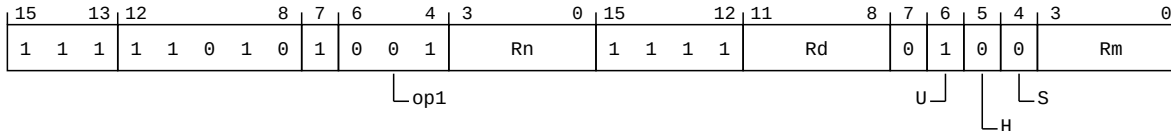
Encoding

UADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

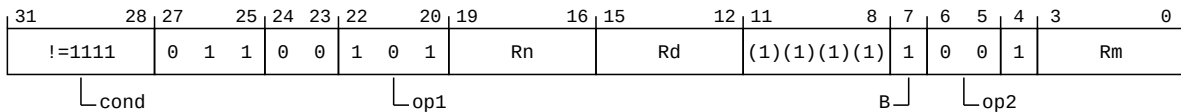
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0] = sum1[15:0];
    R(d)[31:16] = sum2[15:0];
    PSTATE.GE[1:0] = if sum1 >= 0x10000 then '11' else '00';
    PSTATE.GE[3:2] = if sum2 >= 0x10000 then '11' else '00';
end;
```

C5.1.260 UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results of the additions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



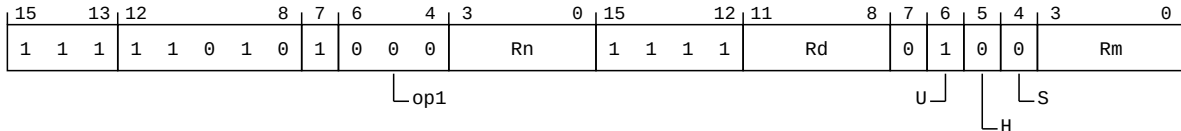
Encoding

UADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

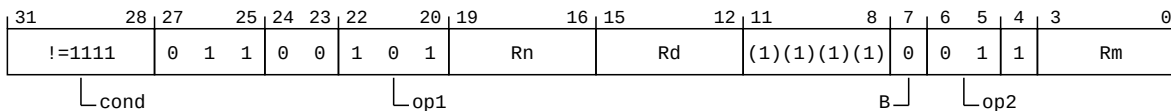
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0] = sum1[7:0];
    R(d)[15:8] = sum2[7:0];
    R(d)[23:16] = sum3[7:0];
    R(d)[31:24] = sum4[7:0];
    PSTATE.GE[0] = if sum1 >= 0x100 then '1' else '0';
    PSTATE.GE[1] = if sum2 >= 0x100 then '1' else '0';
    PSTATE.GE[2] = if sum3 >= 0x100 then '1' else '0';
    PSTATE.GE[3] = if sum4 >= 0x100 then '1' else '0';
end;
```

C5.1.261 UASX

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

UASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

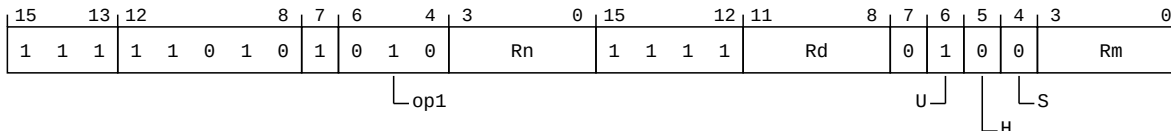
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

UASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

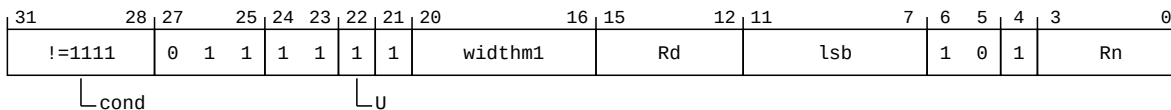
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0] = diff[15:0];
    R(d)[31:16] = sum[15:0];
    PSTATE.GE[1:0] = if diff >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if sum >= 0x10000 then '11' else '00';
end;
```

C5.1.262 UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

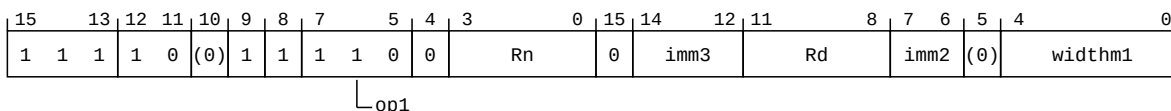
```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(lsb);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let lsbit : integer{} = UInt(imm3::imm2);
let widthminus1 : integer{} = UInt(widthm1);
let msbit : integer{} = lsbit + widthminus1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;
if msbit > 31 then UnpredictableProcedure(); end;
```


CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<lsb>

For the “A1” variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the “lsb” field.

For the “T1” variant: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the “imm3:imm2” field.

<width>

Is the width of the field, in the range 1 to 32-`<lsb>`, encoded in the “widthm1” field as `<width>-1`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R(d) = ZeroExtend{32}(R(n)[msbit:lsbit]);
end;
```

C5.1.263 UDF

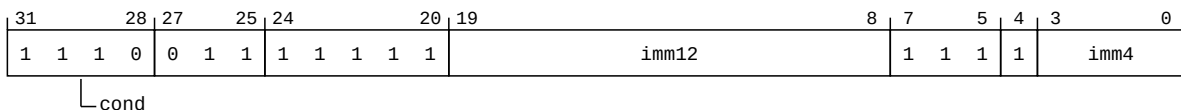
Permanently Undefined generates an Undefined Instruction exception.

The encodings for UDF used in this section are defined as permanently UNDEFINED. However:

- With the T32 instruction set, Arm deprecates using the UDF instruction in an IT block.
- In the A32 instruction set, UDF is not conditional.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



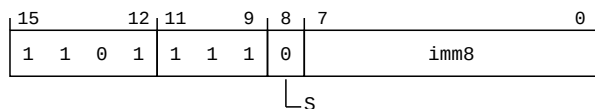
Encoding

UDF{<c>}{<q>}{#}<imm>

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm12::imm4);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T1



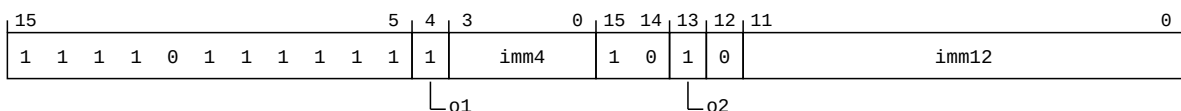
Encoding

UDF{<c>}{<q>}{#}<imm>

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm8);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T2



Encoding

```
UDF{<c>}{<q>} {#}<imm>
```

```
UDF{<c>}.W {#}<imm> // (<imm> can be represented in T1)
```

Decode for this encoding

```
let imm32 : bits(32) = ZeroExtend{}(imm4::imm12);  
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). <c> must be AL or omitted.

For the “T1” and “T2” variants: see [Standard assembler syntax fields](#). Arm deprecates using any <c> value other than AL.

<q>

See [Standard assembler syntax fields](#).

<imm>

For the “A1” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm12:imm4” field. The PE ignores the value of this constant.

For the “T1” variant: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the “imm8” field. The PE ignores the value of this constant.

For the “T2” variant: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm4:imm12” field. The PE ignores the value of this constant.

Operation

```
if ConditionPassed() then  
    EncodingSpecificOperations();  
    Undefined();  
end;
```

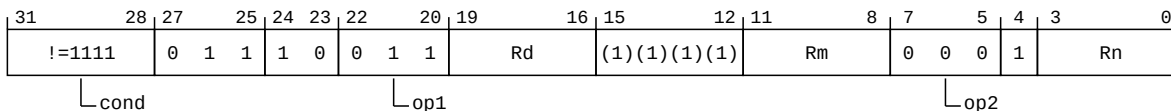
C5.1.264 UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

See [Divide instructions](#) for more information about this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



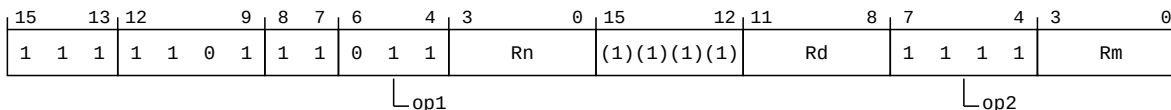
Encoding

UDIV{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UDIV{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register holding the dividend, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the divisor, encoded in the “Rm” field.

Operation

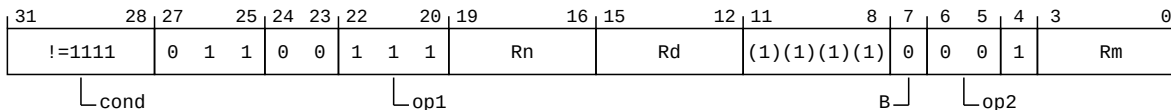
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let dividend : integer = UInt(R(n));
    let divisor : integer = UInt(R(m));
    var result : integer;
    if divisor == 0 then
        result = 0;
    else
        result = dividend DIVRM divisor;
    end;
    R(d) = result[31:0];
end;
```

C5.1.265 UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



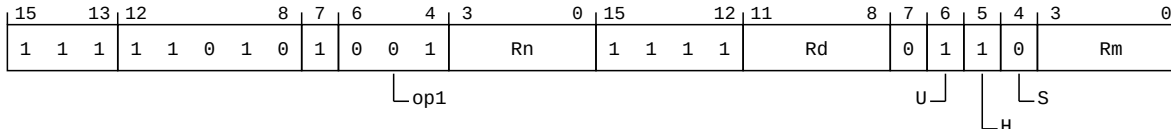
Encoding

UHADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UHADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

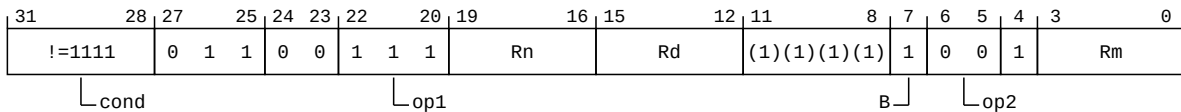
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0] = sum1[16:1];
    R(d)[31:16] = sum2[16:1];
end;
```

C5.1.266 UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



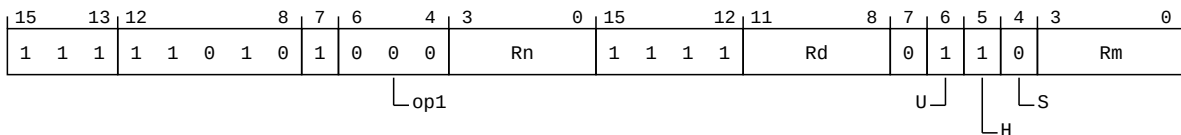
Encoding

UHADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UHADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

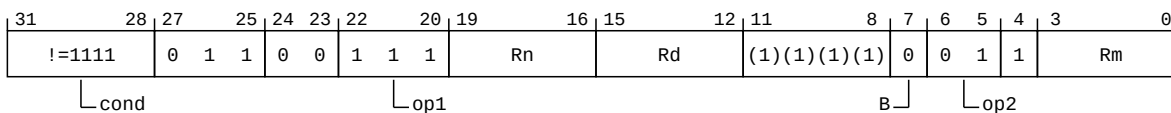
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0]    = sum1[8:1];
    R(d)[15:8]   = sum2[8:1];
    R(d)[23:16]  = sum3[8:1];
    R(d)[31:24]  = sum4[8:1];
end;
```

C5.1.267 UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

UHASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

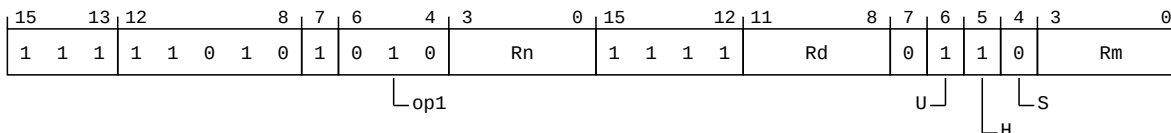
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

UHASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

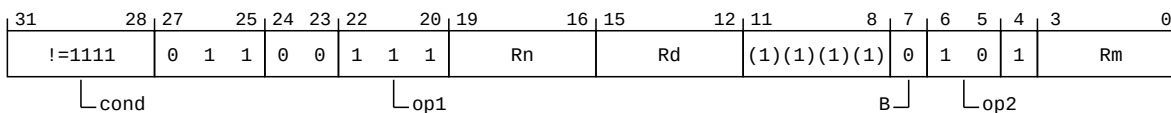
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0] = diff[16:1];
    R(d)[31:16] = sum[16:1];
end;
```

C5.1.268 UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



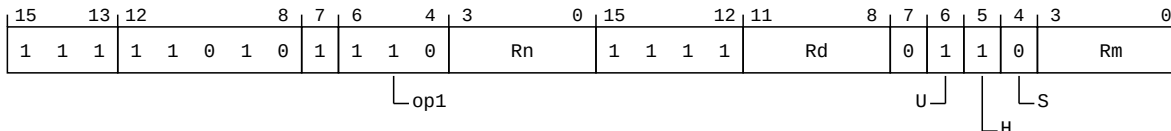
Encoding

UHSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UHSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

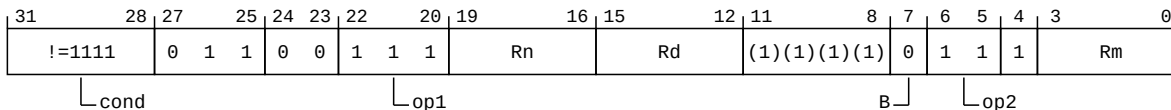
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0] = sum[16:1];
    R(d)[31:16] = diff[16:1];
end;
```

C5.1.269 UHSUB16

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



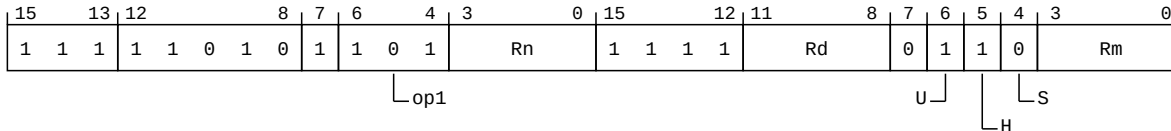
Encoding

UHSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UHSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

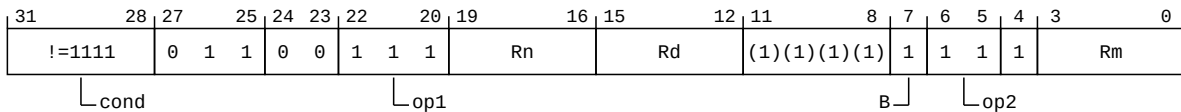
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0] = diff1[16:1];
    R(d)[31:16] = diff2[16:1];
end;
```

C5.1.270 UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



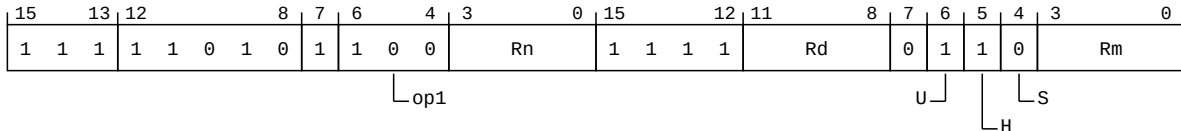
Encoding

UHSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UHSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

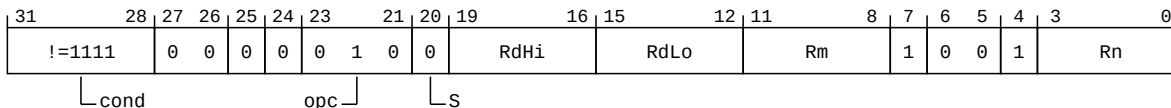
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0]    = diff1[8:1];
    R(d)[15:8]   = diff2[8:1];
    R(d)[23:16]  = diff3[8:1];
    R(d)[31:24]  = diff4[8:1];
end;
```

C5.1.271 UMAAL

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

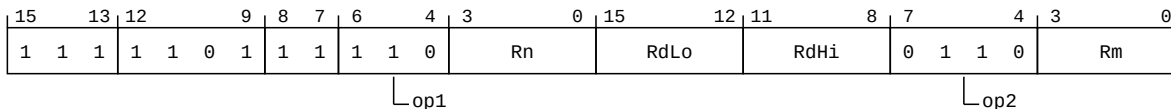
```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $dHi == dLo$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $dHi == dLo$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<RdLo>

Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
let result : integer = UInt(R(n)) * UInt(R(m)) + UInt(R(dHi)) + UInt(R(dLo));
R(dHi) = result[63:32];
R(dLo) = result[31:0];
end;
```

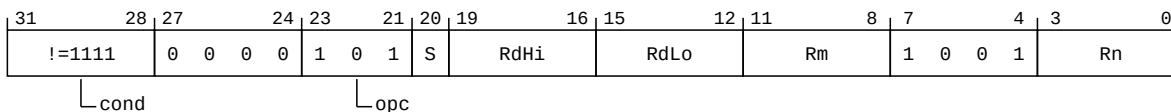
C5.1.272 UMLAL, UMLALS

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Flag setting variant

Applies when (S == 1)

UMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the Not flag setting variant

Applies when $(S == 0)$

UMLAL{<c>}{<q>}<RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```

let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;

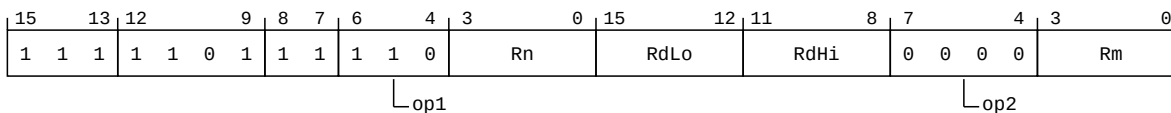
```

CONSTRAINED UNPREDICTABLE behavior

If $d_{Hi} == d_{Lo}$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

T1



Encoding

UMLAL{<c>}{<q>}<RdLo>,<RdHi>,<Rn>,<Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<RdLo>

Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = UInt(R(n)) * UInt(R(m)) + UInt(R(dHi)::R(dLo));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

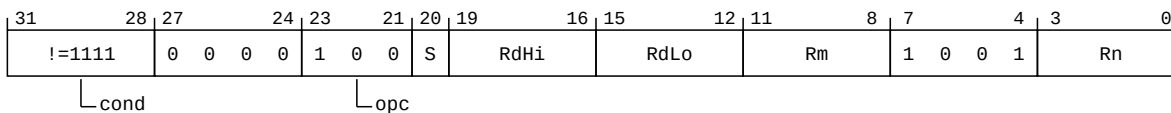
C5.1.273 UMULL, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Flag setting variant

Applies when (S == 1)

UMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Encoding for the Not flag setting variant

Applies when $(S == 0)$

UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for all variants of this encoding

```

let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
if dHi == dLo then UnpredictableProcedure(); end;

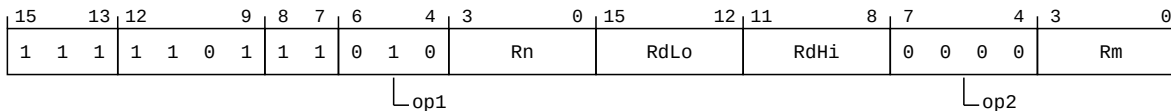
```

CONSTRAINED UNPREDICTABLE behavior

If $d_{Hi} == d_{Lo}$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

T1



Encoding

UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

Decode for this encoding

```
let dLo : integer = UInt(RdLo);
let dHi : integer = UInt(RdHi);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let setflags : boolean = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<RdLo>

Is the general-purpose destination register for the lower 32 bits of the result, encoded in the “RdLo” field.

<RdHi>

Is the general-purpose destination register for the upper 32 bits of the result, encoded in the “RdHi” field.

<Rn>

Is the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

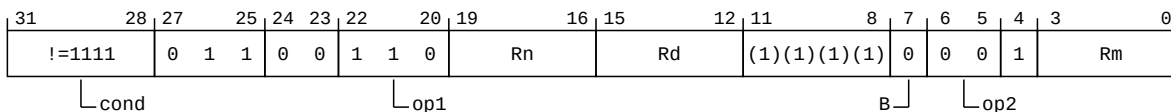
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let result : integer = UInt(R(n)) * UInt(R(m));
    R(dHi) = result[63:32];
    R(dLo) = result[31:0];
    if setflags then
        PSTATE.N = result[63];
        PSTATE.Z = IsZeroBit{64}(result[63:0]);
        // PSTATE.C, PSTATE.V unchanged
    end;
end;
```

C5.1.274 UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



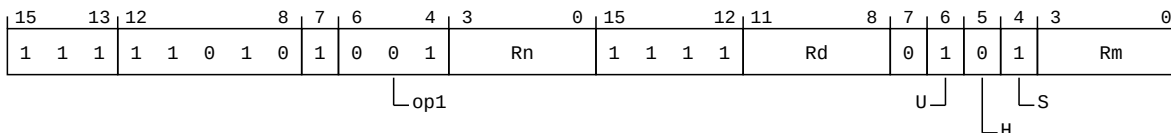
Encoding

UQADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UQADD16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

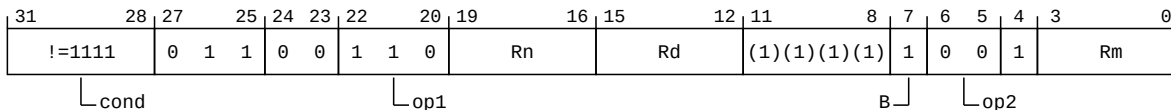
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[15:0]) + UInt(R(m)[15:0]);
    let sum2 : integer = UInt(R(n)[31:16]) + UInt(R(m)[31:16]);
    R(d)[15:0] = UnsignedSat{16}(sum1);
    R(d)[31:16] = UnsignedSat{16}(sum2);
end;
```

C5.1.275 UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



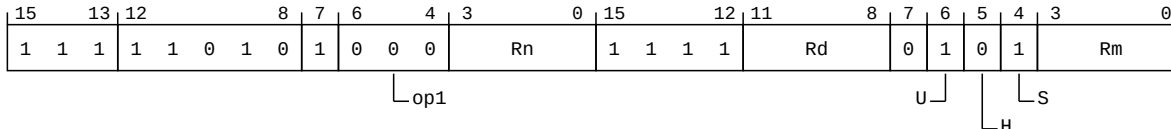
Encoding

UQADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UQADD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

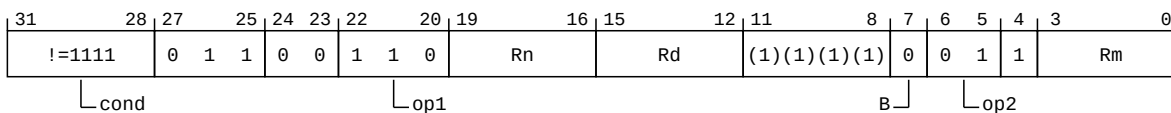
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum1 : integer = UInt(R(n)[7:0]) + UInt(R(m)[7:0]);
    let sum2 : integer = UInt(R(n)[15:8]) + UInt(R(m)[15:8]);
    let sum3 : integer = UInt(R(n)[23:16]) + UInt(R(m)[23:16]);
    let sum4 : integer = UInt(R(n)[31:24]) + UInt(R(m)[31:24]);
    R(d)[7:0] = UnsignedSat{8}(sum1);
    R(d)[15:8] = UnsignedSat{8}(sum2);
    R(d)[23:16] = UnsignedSat{8}(sum3);
    R(d)[31:24] = UnsignedSat{8}(sum4);
end;
```

C5.1.276 UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



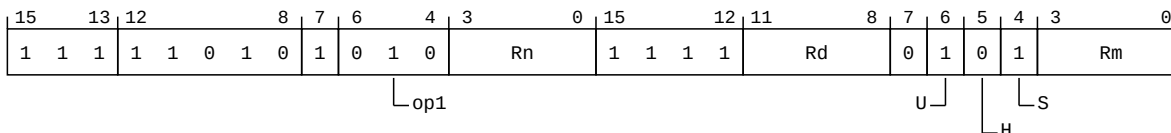
Encoding

UQASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UQASX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

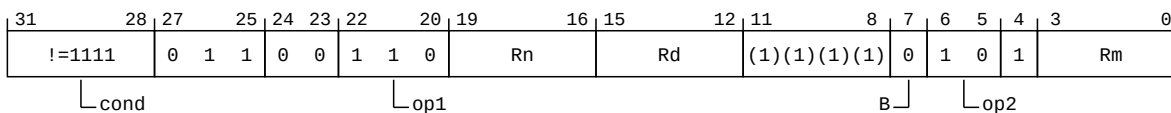
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff : integer = UInt(R(n)[15:0]) - UInt(R(m)[31:16]);
    let sum : integer = UInt(R(n)[31:16]) + UInt(R(m)[15:0]);
    R(d)[15:0] = UnsignedSat{16}(diff);
    R(d)[31:16] = UnsignedSat{16}(sum);
end;
```

C5.1.277 UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



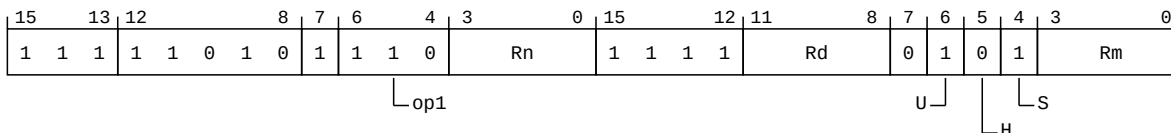
Encoding

UQSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UQSAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

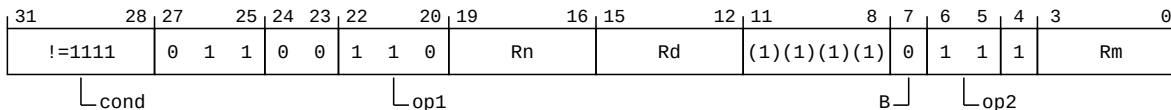
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0] = UnsignedSat{16}(sum);
    R(d)[31:16] = UnsignedSat{16}(diff);
end;
```

C5.1.278 UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



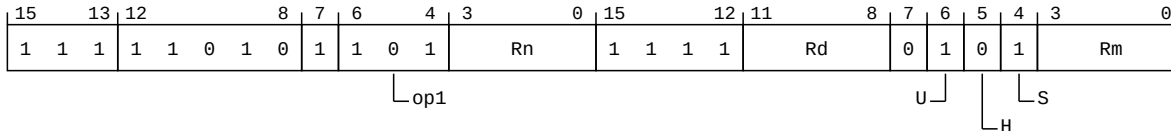
Encoding

UQSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UQSUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

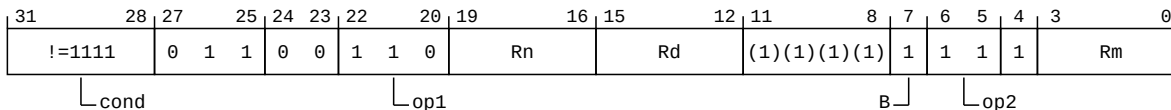
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0] = UnsignedSat{16}(diff1);
    R(d)[31:16] = UnsignedSat{16}(diff2);
end;
```

C5.1.279 UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



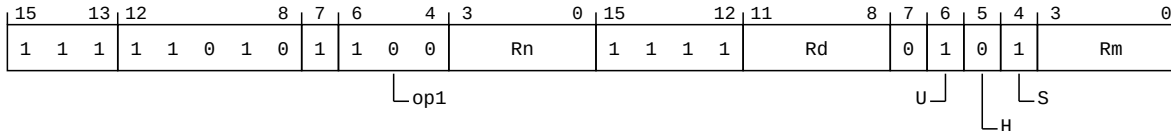
Encoding

UQSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UQSUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0] = UnsignedSat{8}(diff1);
    R(d)[15:8] = UnsignedSat{8}(diff2);
    R(d)[23:16] = UnsignedSat{8}(diff3);
    R(d)[31:24] = UnsignedSat{8}(diff4);
end;
```

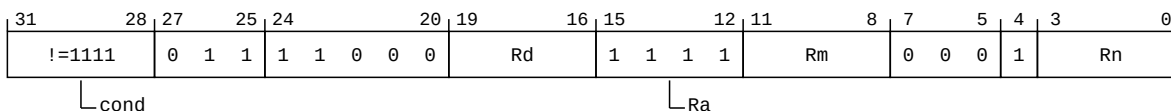
C5.1.280 USAD8

Unsigned Sum of Absolute Differences

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



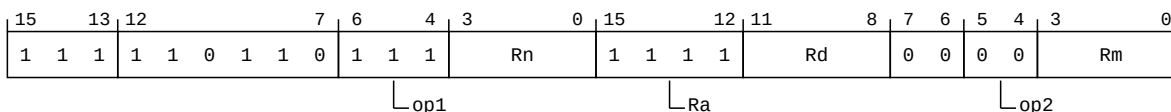
Encoding

USAD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

USAD8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let absdiff1 : integer = Abs(UInt(R(n)[7:0]) - UInt(R(m)[7:0]));
    let absdiff2 : integer = Abs(UInt(R(n)[15:8]) - UInt(R(m)[15:8]));
    let absdiff3 : integer = Abs(UInt(R(n)[23:16]) - UInt(R(m)[23:16]));
    let absdiff4 : integer = Abs(UInt(R(n)[31:24]) - UInt(R(m)[31:24]));
    let result : integer = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R(d) = result[31:0];
end;
```

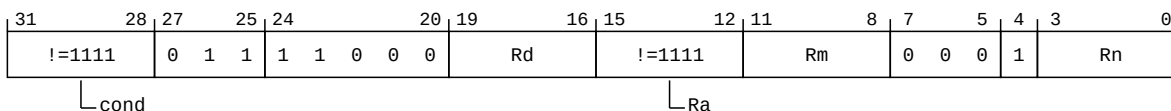
C5.1.281 USADA8

Unsigned Sum of Absolute Differences and Accumulate

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

USDA8{<c>}{<q>}<Rd>,<Rn>,<Rm>,<Ra>

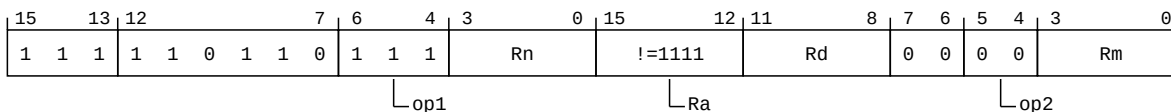
Decode for this encoding

```

if Ra == '1111' then See("USAD8"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

USDA8 { <c> } { <q> } <Rd> , <Rn> , <Rm> , <Ra>

Decode for this encoding

```

if Ra == '1111' then See("USAD8"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let a : integer = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<Ra>

Is the third general-purpose source register holding the addend, encoded in the “Ra” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let absdiff1 : integer = Abs(UInt(R(n)[7:0]) - UInt(R(m)[7:0]));
    let absdiff2 : integer = Abs(UInt(R(n)[15:8]) - UInt(R(m)[15:8]));
    let absdiff3 : integer = Abs(UInt(R(n)[23:16]) - UInt(R(m)[23:16]));
    let absdiff4 : integer = Abs(UInt(R(n)[31:24]) - UInt(R(m)[31:24]));
    let result : integer = UInt(R(a)) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R(d) = result[31:0];
end;
```

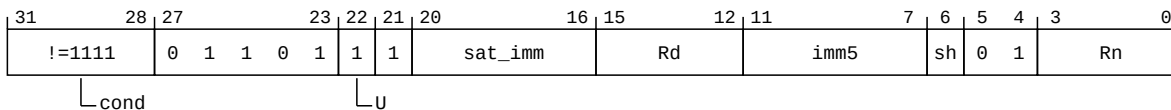
C5.1.282 USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

This instruction sets [PSTATE.Q](#) to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Arithmetic shift right variant

Applies when (sh == 1)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Encoding for the Logical shift left variant

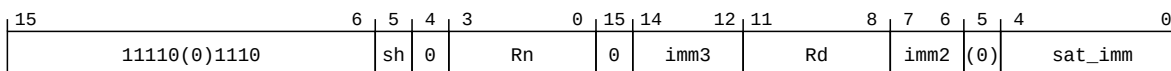
Applies when (sh == 0)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

Decode for all variants of this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(sh::'0', imm5);
if d == 15 || n == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the Arithmetic shift right variant

Applies when (sh == 1 && !(imm3 == 000 && imm2 == 00))

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Encoding for the Logical shift left variant

Applies when (sh == 0)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

Decode for all variants of this encoding

```

if sh == '1' && (imm3::imm2) == '00000' then See("USAT16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
let (shift_t, shift_n) : (SRTType, integer) = DecodeImmShift(sh::'0', imm3::imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the bit position for saturation, in the range 0 to 31, encoded in the “sat_imm” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

<amount>

For the “A1 Arithmetic shift right” variant: is the shift amount, in the range 1 to 32 encoded in the “imm5” field as <amount> modulo 32.

For the “A1 Logical shift left” variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm5” field.

For the “T1 Arithmetic shift right” variant: is the shift amount, in the range 1 to 31 encoded in the “imm3:imm2” field as <amount>.

For the “T1 Logical shift left” variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm3:imm2” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    let operand : bits(32) = Shift{}(R(n), shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
    let (result, sat) : (bits(saturate_to), boolean) = UnsignedSatQ{saturate_to}(SInt(operand));
    R(d) = ZeroExtend{32}(result);
    if sat then
        PSTATE.Q = '1';
    end;
end;

```

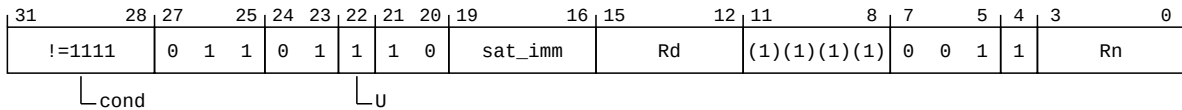
C5.1.283 USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.

This instruction sets **PSTATE.Q** to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

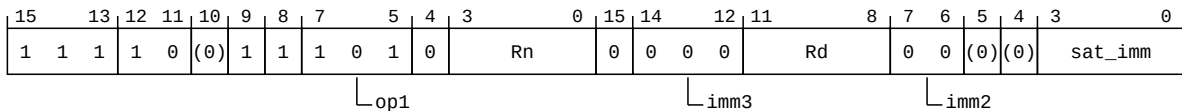
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
if d == 15 || n == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let saturate_to : integer{} = UInt(sat_imm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UnpredictableProcedure(); end;

```

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<imm>

Is the bit position for saturation, in the range 0 to 15, encoded in the “sat_imm” field.

<Rn>

Is the general-purpose source register, encoded in the “Rn” field.

Operation

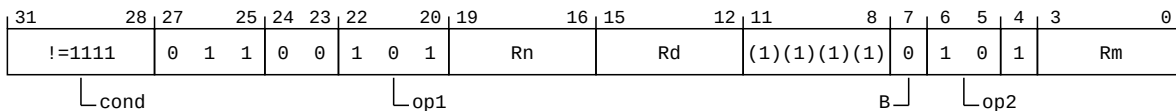
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let operand1 = SInt(R(n)[15:0]);
    let operand2 = SInt(R(n)[31:16]);
    let (result1, sat1) : (bits(saturate_to), boolean) = UnsignedSatQ(saturate_to)(operand1);
    let (result2, sat2) : (bits(saturate_to), boolean) = UnsignedSatQ(saturate_to)(operand2);
    R(d)[15:0] = ZeroExtend{16}(result1);
    R(d)[31:16] = ZeroExtend{16}(result2);
    if sat1 || sat2 then
        PSTATE.Q = '1';
    end;
end;
```

C5.1.284 USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



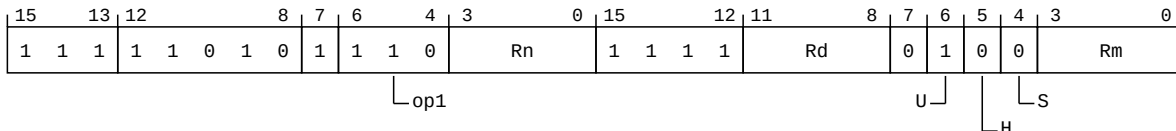
Encoding

USAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

USAX{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

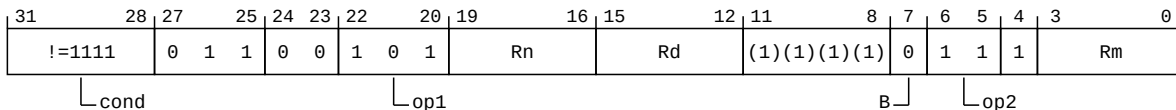
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let sum : integer = UInt(R(n)[15:0]) + UInt(R(m)[31:16]);
    let diff : integer = UInt(R(n)[31:16]) - UInt(R(m)[15:0]);
    R(d)[15:0] = sum[15:0];
    R(d)[31:16] = diff[15:0];
    PSTATE.GE[1:0] = if sum >= 0x10000 then '11' else '00';
    PSTATE.GE[3:2] = if diff >= 0 then '11' else '00';
end;
```

C5.1.285 USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results of the subtractions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



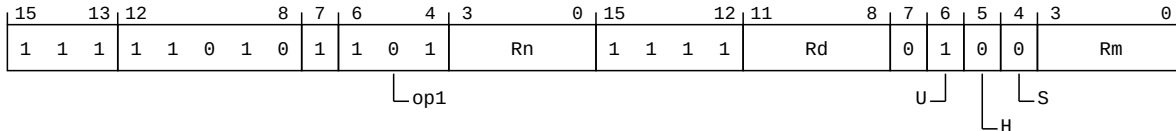
Encoding

USUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

USUB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

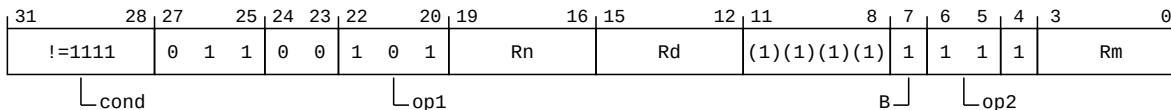
```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[15:0]) - UInt(R(m)[15:0]);
    let diff2 : integer = UInt(R(n)[31:16]) - UInt(R(m)[31:16]);
    R(d)[15:0] = diff1[15:0];
    R(d)[31:16] = diff2[15:0];
    PSTATE.GE[1:0] = if diff1 >= 0 then '11' else '00';
    PSTATE.GE[3:2] = if diff2 >= 0 then '11' else '00';
end;
```

C5.1.286 USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets [PSTATE.GE](#) according to the results of the subtractions.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

USUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

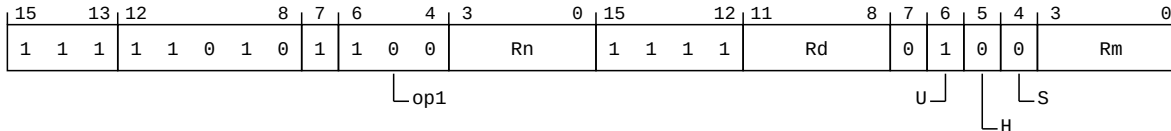
Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

USUB8{<c>}{<q>} {<Rd>, }<Rn>, <Rm>

Decode for this encoding

```

let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UnpredictableProcedure(); end;

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let diff1 : integer = UInt(R(n)[7:0]) - UInt(R(m)[7:0]);
    let diff2 : integer = UInt(R(n)[15:8]) - UInt(R(m)[15:8]);
    let diff3 : integer = UInt(R(n)[23:16]) - UInt(R(m)[23:16]);
    let diff4 : integer = UInt(R(n)[31:24]) - UInt(R(m)[31:24]);
    R(d)[7:0] = diff1[7:0];
    R(d)[15:8] = diff2[7:0];
    R(d)[23:16] = diff3[7:0];
    R(d)[31:24] = diff4[7:0];
    PSTATE.GE[0] = if diff1 >= 0 then '1' else '0';
    PSTATE.GE[1] = if diff2 >= 0 then '1' else '0';
    PSTATE.GE[2] = if diff3 >= 0 then '1' else '0';
    PSTATE.GE[3] = if diff4 >= 0 then '1' else '0';
end;
```

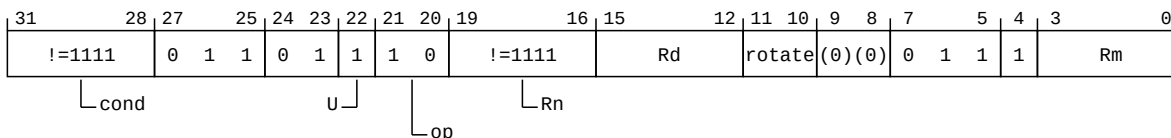
C5.1.287 UXTAB

Unsigned Extend and Add Byte

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



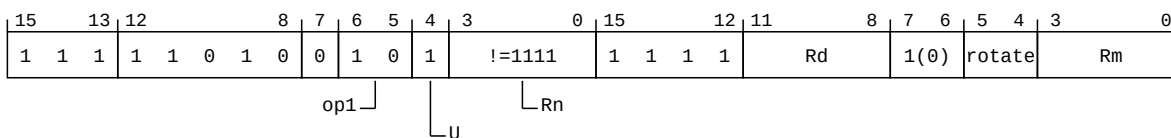
Encoding

UXTAB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UXTAB{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTB"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + ZeroExtend{32}(rotated[7:0]);
end;
```

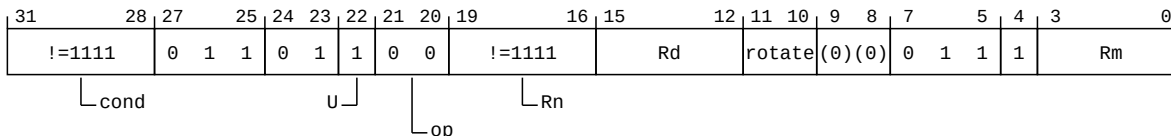
C5.1.288 UXTAB16

Unsigned Extend and Add Byte 16

Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



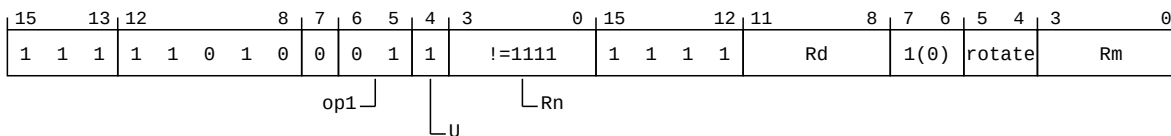
Encoding

```
UXTAB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("UXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
UXTAB16{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then See("UXTB16"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0] = R(n)[15:0] + ZeroExtend{16}(rotated[7:0]);
    R(d)[31:16] = R(n)[31:16] + ZeroExtend{16}(rotated[23:16]);
end;
```

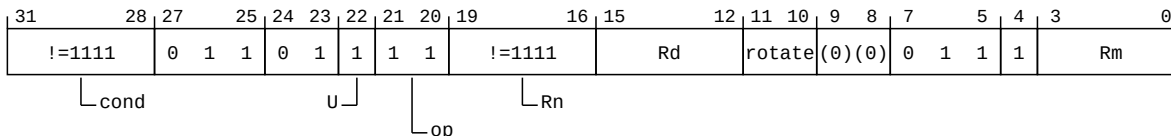
C5.1.289 UXTAH

Unsigned Extend and Add Halfword

Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



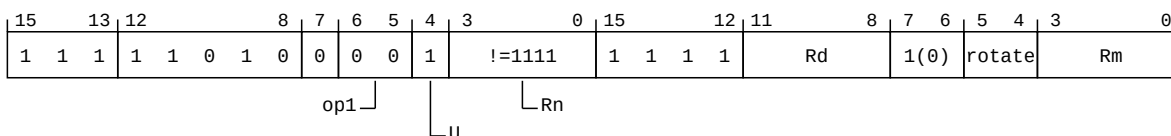
Encoding

UXTAH{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

UXTAH{<c>}{<q>} {<Rd>, }<Rn>, <Rm> {, ROR #<amount>}

Decode for this encoding

```
if Rn == '1111' then See("UXTH"); end;
let d : integer = UInt(Rd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rn>

Is the first general-purpose source register, encoded in the “Rn” field.

<Rm>

Is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = R(n) + ZeroExtend{32}(rotated[15:0]);
end;
```

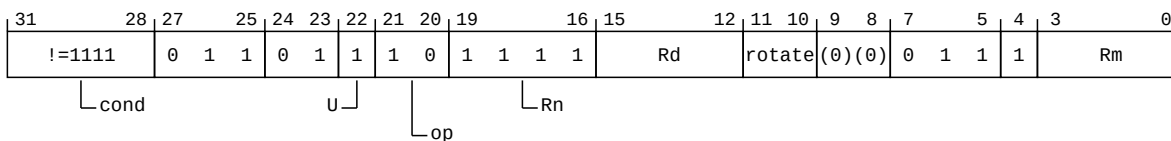
C5.1.290 UXTB

Unsigned Extend Byte

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



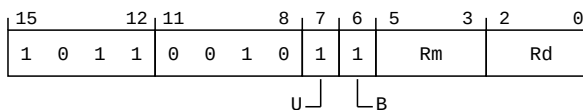
Encoding

```
UXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



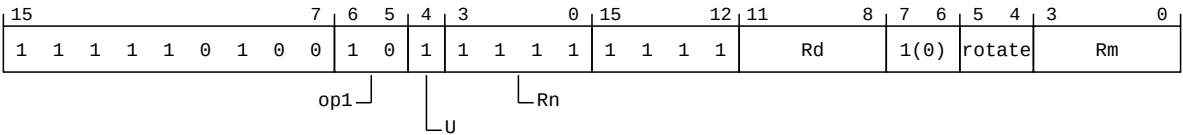
Encoding

```
UXTB{<c>}{<q>} {<Rd>, }<Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```


T2



Encoding

```
UXTB{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}
```

```
UXTB{<c>}.W {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = ZeroExtend{32}(rotated[7:0]);
end;
```

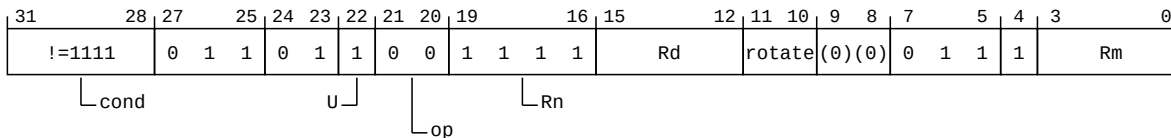
C5.1.291 UXTB16

Unsigned Extend Byte 16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



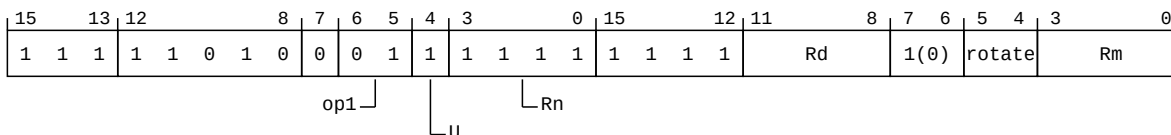
Encoding

```
UXTB16{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
UXTB16{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

For the “A1” variant: is the general-purpose source register, encoded in the “Rm” field.

For the “T1” variant: is the second general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d)[15:0] = ZeroExtend{16}(rotated[7:0]);
    R(d)[31:16] = ZeroExtend{16}(rotated[23:16]);
end;
```

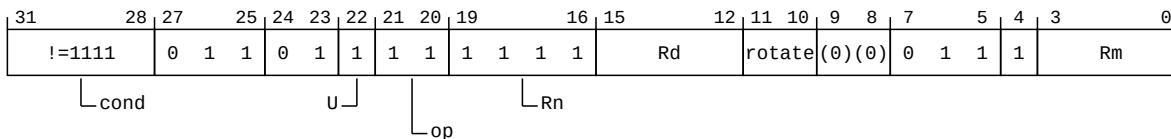
C5.1.292 UXTH

Unsigned Extend Halfword

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



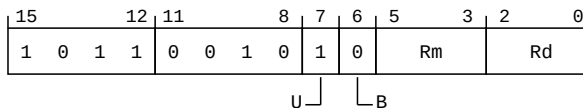
Encoding

```
UXTH{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = UInt(rotate::'000');
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

T1



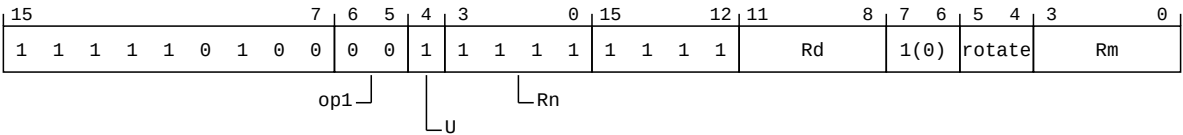
Encoding

```
UXTH{<c>}{<q>} {<Rd>, }<Rm>
```

Decode for this encoding

```
let d : integer = UInt(Rd);
let m : integer = UInt(Rm);
let rotation : integer = 0;
```

T2



Encoding

```
UXTH{<c>}{<q>} {<Rd>, }<Rm> {, ROR #<amount>}  
  
UXTH{<c>}.W {<Rd>, }<Rm> // (<Rd>, <Rm> can be represented in T1)
```

Decode for this encoding

```
let d : integer = UInt(Rd);  
let m : integer = UInt(Rm);  
let rotation : integer = UInt(rotate::'000');  
// Armv8-A removes UNPREDICTABLE for R13  
if d == 15 || m == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rd>

Is the general-purpose destination register, encoded in the “Rd” field.

<Rm>

Is the general-purpose source register, encoded in the “Rm” field.

<amount>

Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	[absent]
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    let rotated : bits(32) = ROR(R(m), rotation);
    R(d) = ZeroExtend{32}(rotated[15:0]);
end;
```

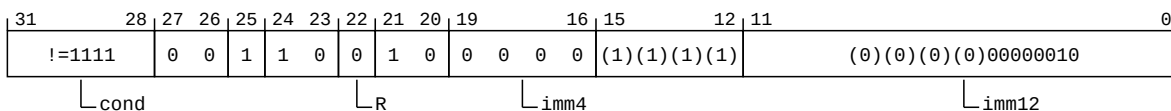
C5.1.293 WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see [Wait For Event and Send Event](#).

As described in [Wait For Event and Send Event](#), the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see [HCR.TWE](#), [SCR.TWE](#), and [SCTLR.nTWE](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



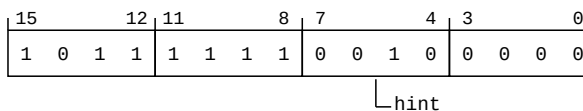
Encoding

WFE{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1



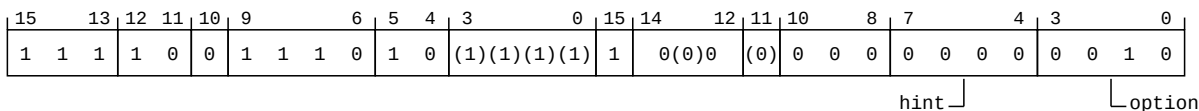
Encoding

WFE{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2



Encoding

WFE{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
if IsEventRegisterSet() then
    ClearEventRegister();
else
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS.
        AArch32_CheckForWfxTrap(EL1, WfxType_WFE);
    end;
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for traps described by the Hypervisor.
        AArch32_CheckForWfxTrap(EL2, WfxType_WFE);
    end;
    WaitForEvent();
end;
end;
```

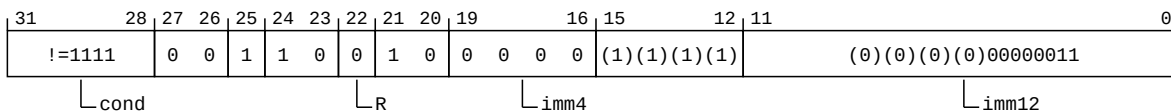
C5.1.294 WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see [Wait For Interrupt](#).

As described in [Wait For Interrupt](#), the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see [HCR.TWI](#), [SCR.TWI](#), and [SCTLR.nTWI](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



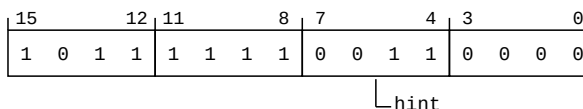
Encoding

`WFI{<c>}{<q>}`

Decode for this encoding

// No additional decoding required

T1



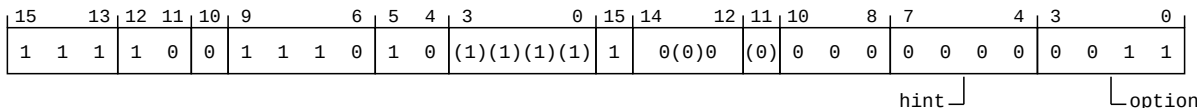
Encoding

`WFI{<c>}{<q>}`

Decode for this encoding

// No additional decoding required

T2



Encoding

`WFI{<c>}.W`

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !AArch32_InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32_CheckForWfxTrap(EL1, WfxType_WFI);
        end;
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
            // Check for traps described by the Hypervisor.
            AArch32_CheckForWfxTrap(EL2, WfxType_WFI);
        end;
        WaitForInterrupt();
    end;
end;
```

C5.1.295 YIELD

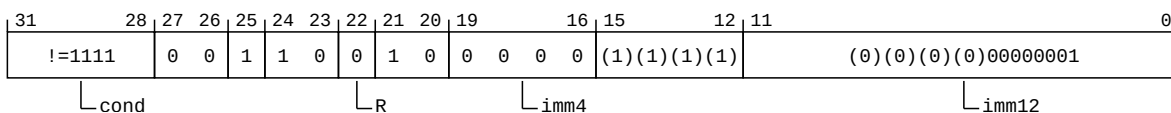
Yield hint

YIELD is a hint instruction. Software with a multithreading capability can use a **YIELD** instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction see [The Yield instruction](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1



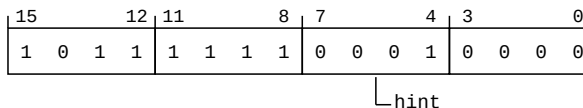
Encoding

YIELD{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T1



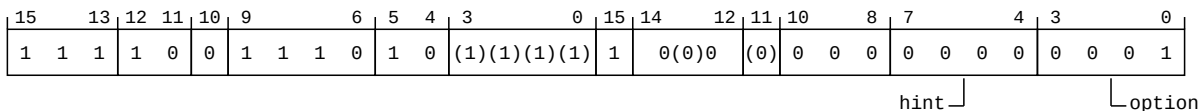
Encoding

YIELD{<c>}{<q>}

Decode for this encoding

// No additional decoding required

T2



Encoding

YIELD{<c>}.W

Decode for this encoding

// No additional decoding required

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
end;
```

C5.2 Encoding and use of banked register transfer instructions

Software executing at EL1 or higher can use the `MRS` (banked register) and `MSR` (banked register) instructions to transfer values between the general-purpose registers and Special-purpose registers. One particular use of these instructions is for a hypervisor to save or restore the register values of a Guest OS. The following sections give more information about these instructions:

- [Register arguments in the banked register transfer instructions.](#)
- [Usage restrictions on the banked register transfer instructions.](#)
- [Encoding the register argument in the banked register transfer instructions.](#)
- [Pseudocode support for the banked register transfer instructions.](#)

For descriptions of the instructions see [MRS \(Banked register\)](#) and [MSR \(Banked register\)](#).

C5.2.1 Register arguments in the banked register transfer instructions

Figure C5-1 shows the banked general-purpose registers and Special-purpose registers:

		Associated PE mode							
		User or System	Hyp	Supervisor	Abort	Undefined	Monitor	IRQ	FIQ
General-purpose registers	R8_usr								R8_fiq
	R9_usr								R9_fiq
	R10_usr								R10_fiq
	R11_usr								R11_fiq
	R12_usr								R12_fiq
	SP_usr	SP_usr	SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
Special-purpose registers	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
		SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq	
		ELR_hyp							

For the general-purpose registers, if no other register is shown, the *current mode register* is the `_usr` register. So, for example, the full set of current mode registers, including the registers that are not banked:

- For Hyp mode, is {R0_usr - R12_usr, SP_hyp, LR_usr, SPSR_hyp, ELR_hyp}.
- For Abort mode, is {R0_usr - R12_usr, SP_abt, LR_abt, SPSR_abt}.

Figure C5-1 Banking of general-purpose and Special-purpose registers

Note

- System mode uses the same set of registers as User mode. Neither of these modes can access an [SPSR](#), except that System mode can use the `MRS` (banked register) and `MSR` (banked register) instructions to access some [SPSRs](#), as described in [Usage restrictions on the banked register transfer instructions](#).
- General-purpose registers R0-R7, that are not banked, cannot be accessed using the `MRS` (banked register) and `MSR` (banked register) instructions.
- In addition to the registers shown in [Figure C5-1](#), the [DLR](#) and [DSPSR](#) are AArch32 System registers that map onto the AArch64 Special-purpose registers `DLR_EL0` and `DSPSR_EL0`. However, [DLR](#) and [DSPSR](#) are not accessible using the `MRS` (banked register) and `MSR` (banked register) instructions.

Software using an `MRS` (banked register) or `MSR` (banked register) instruction specifies one of these registers using a name shown in [Figure C5-1](#), or an alternative name for SP or LR. These registers can be grouped as follows:

R8-R12	Each of these registers has two banked copies, <code>_usr</code> and <code>_fiq</code> , for example R8_usr and R8_fiq.
SP	There is a banked copy of SP for every mode except System mode. For example, SP_svc is the SP for Supervisor mode.
LR	There is a banked copy of LR for every mode except System mode and Hyp mode. For example, LR_svc is the LR for Supervisor mode.

SPSR	There is a banked copy of SPSR for every mode except System mode and User mode.
ELR_hyp	Except for the operations provided by MRS (banked register) and MSR (banked register) , ELR_hyp is accessible only from Hyp mode. It is not banked.

C5.2.2 Usage restrictions on the banked register transfer instructions

[MRS \(banked register\)](#) and [MSR \(banked register\)](#) instructions are **CONSTRAINED UNPREDICTABLE** if any of the following applies:

- The instruction is executed in User mode.
- The instruction accesses a banked register that is not implemented, or that either:
 - Is not accessible from the current Privilege level and Security state.
 - Can be accessed from the current mode by using a different instruction.

[MSR \(banked register\)](#) and [MRS \(banked register\)](#) describes the permitted **CONSTRAINED UNPREDICTABLE** behavior.

An [MRS \(banked register\)](#) instruction or an [MSR \(banked register\)](#) instruction executed:

- At Non-secure EL1 cannot access any Hyp mode banked registers.
- At Non-secure EL1 or EL2 cannot access any Monitor mode banked registers.
- In a Secure mode other than Monitor mode cannot access any Hyp banked registers.

This means that the banked registers that [MRS \(banked register\)](#) and [MSR \(banked register\)](#) instructions cannot access are:

From Hyp mode

- The current mode registers R8_usr-R12_usr, SP_hyp, LR_usr, and SPSR_hyp.

Note

[MRS \(banked register\)](#) and [MSR \(banked register\)](#) instructions can access the current mode register [ELR_hyp](#).

From FIQ mode

- The Hyp mode registers SP_hyp, SPSR_hyp, and [ELR_hyp](#).
- The current mode registers R8_fiq-R12_fiq, SP_fiq, LR_fiq, and SPSR_fiq.

From System mode

- The Hyp mode registers SP_hyp, SPSR_hyp, and [ELR_hyp](#).
- The current mode registers R8_usr-R12_usr, SP_usr, and LR_usr.

From Supervisor mode, Abort mode, Undefined mode, and IRQ mode

- The Hyp mode registers SP_hyp, SPSR_hyp, and [ELR_hyp](#).
- The current mode registers R8_usr-R12_usr, SP_<current_mode>, LR_<current_mode>, and SPSR_<current_mode>.

For more information, see:

- [Encoding the register argument in the banked register transfer instructions.](#)
- [Pseudocode support for the banked register transfer instructions.](#)
- [MRS \(Banked register\).](#)
- [MSR \(Banked register\).](#)

Note

CONSTRAINED UNPREDICTABLE behavior must not give access to registers that are not accessible from the current Privilege level and Security state.

C5.2.3 Encoding the register argument in the banked register transfer instructions

The `MRS` (banked register) and `MSR` (banked register) instructions include a 5-bit field, `SYSm`, and an `R` bit, that together encode the register argument for the instruction.

When the `R` bit is set to 0, the argument is a register other than a banked copy of the `SPSR`, and [Table C5-119](#) shows how the `SYSm` field defines the required register argument.

Table C5-119 Banked register encodings when `R==0`

SYSm<2:0>	SYSm<4:3>			
	0b00	0b01	0b10	0b11
0b000	R8_usr	R8_fiq	LR_irq	CONSTRAINED UNPREDICTABLE
0b001	R9_usr	R9_fiq	SP_irq	CONSTRAINED UNPREDICTABLE
0b010	R10_usr	R10_fiq	LR_svc	CONSTRAINED UNPREDICTABLE
0b011	R11_usr	R11_fiq	SP_svc	CONSTRAINED UNPREDICTABLE
0b100	R12_usr	R12_fiq	LR_abt	LR_mon
0b101	SP_usr	SP_fiq	SP_abt	SP_mon
0b110	LR_usr	LR_fiq	LR_und	ELR_hyp
0b111	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	SP_und	SP_hyp

When the `R` bit is set to 1, the argument is a banked copy of the `SPSR`, and [Table C5-120](#) shows how the `SYSm` field defines the required register argument.

Table C5-120 Banked register encodings when `R==1`

SYSm<2:0>	SYSm<4:3>			
	0b00	0b01	0b10	0b11
0b000	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	SPSR_irq	CONSTRAINED UNPREDICTABLE
0b001	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE
0b010	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	SPSR_svc	CONSTRAINED UNPREDICTABLE
0b011	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE
0b100	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	SPSR_abt	SPSR_mon
0b101	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE
0b110	CONSTRAINED UNPREDICTABLE	SPSR_fiq	SPSR_und	SPSR_hyp
0b111	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE	CONSTRAINED UNPREDICTABLE

C5.2.4 Pseudocode support for the banked register transfer instructions

The pseudocode functions `BankedRegisterAccessValid()` and `SPSRaccessValid()` check the validity of `MRS` (banked register) and `MSR` (banked register) accesses. That is, they filter the accesses that are `CONSTRAINED UNPREDICTABLE` either because:

- They attempt to access a register that [Usage restrictions on the banked register transfer instructions](#) shows is not accessible.
- They use an `SYSm<4:0>` encoding that [Encoding the register argument in the banked register transfer instructions](#) shows as `CONSTRAINED UNPREDICTABLE`.

`BankedRegisterAccessValid()` applies to accesses to the banked general-purpose registers, or to `ELR_hyp`, and `SPSRaccessValid()` applies to accesses to the `SPSRs`.

Chapter C6

T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes each instruction. It contains the following sections:

- [Alphabetical list of Advanced SIMD and floating-point instructions](#)

Note

Some headings in this chapter use the term *floating-point register*. This is an abbreviated description, and means a register in the Advanced SIMD and floating-point register file.

C6.1 Alphabetical list of Advanced SIMD and floating-point instructions

This section lists every Advanced SIMD and floating-point instruction in the T32 and A32 instruction sets. For details of the format used see [Format of instruction descriptions](#).

This section is formatted so that each instruction description starts on a new page.

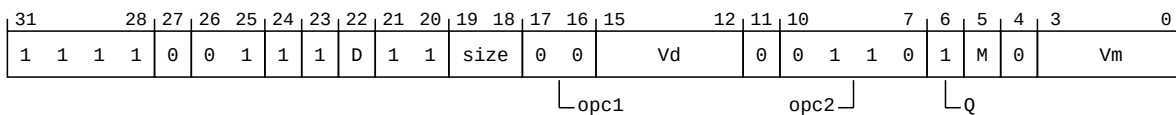
C6.1.1 AESD

AES single round decryption.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AES)



Encoding

AESD.<dt> <Qd>, <Qm>

Decode for this encoding

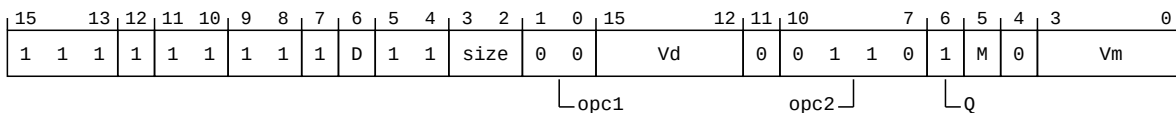
```

if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_AES)



Encoding

AESD.<dt> <Qd>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    let operand1 : bits(128) = Q(d>>1);
    let operand2 : bits(128) = Q(m>>1);
    Q(d>>1) = AESInvSubBytes(AESInvShiftRows(operand1 XOR operand2));
end;
```

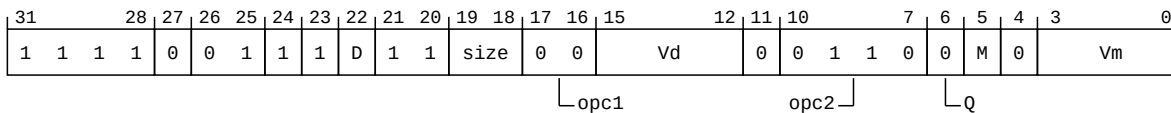
C6.1.2 AESE

AES single round encryption.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AES)



Encoding

AESE.<dt> <Qd>, <Qm>

Decode for this encoding

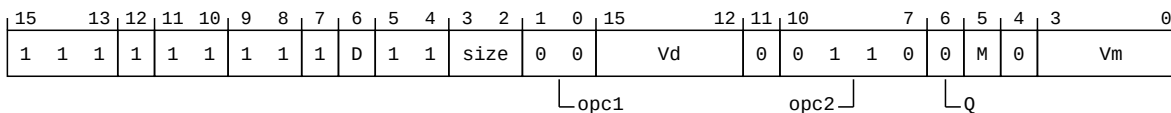
```

if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_AES)



Encoding

AESE.<dt> <Qd>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    let operand1 : bits(128) = Q(d>>1);
    let operand2 : bits(128) = Q(m>>1);
    Q(d>>1) = AESSubBytes(AESShiftRows(operand1 XOR operand2));
end;
```

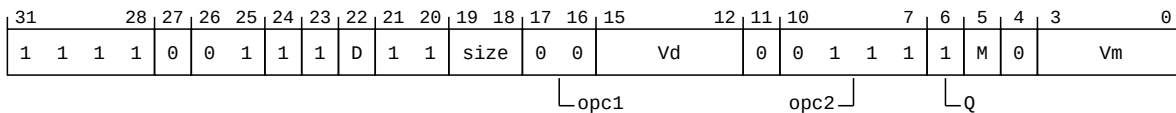
C6.1.3 AESIMC

AES inverse mix columns.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AES)



Encoding

AESIMC.<dt> <Qd>, <Qm>

Decode for this encoding

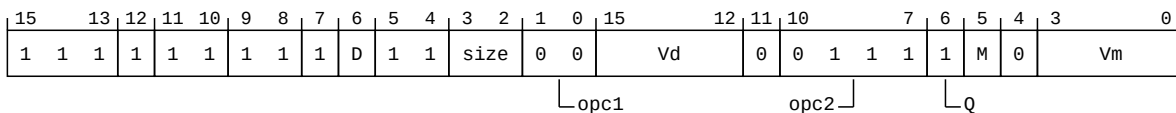
```

if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_AES)



Encoding

AESIMC.<dt> <Qd>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```


Assembler Symbols

<dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q(d>>1) = AESInvMixColumns(Q(m>>1));
end;
```

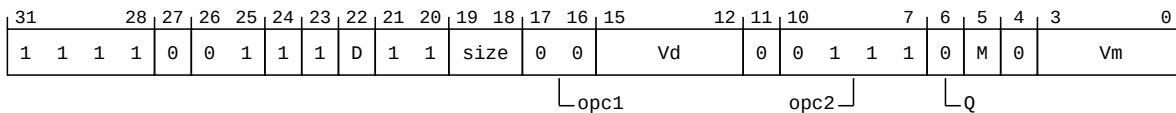
C6.1.4 AESMC

AES mix columns.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AES)



Encoding

AESMC.<dt> <Qd>, <Qm>

Decode for this encoding

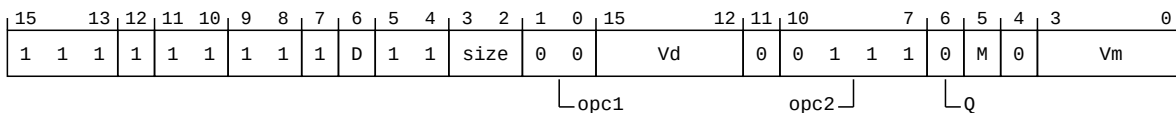
```

if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_AES)



Encoding

AESMC.<dt> <Qd>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_AES) then Undefined(); end;
if size != '00' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<dt>

Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q(d>>1) = AESMixColumns(Q(m>>1));
end;
```

C6.1.5 FLDM*X (FLDMDBX, FLDMIAX)

FLDM*X

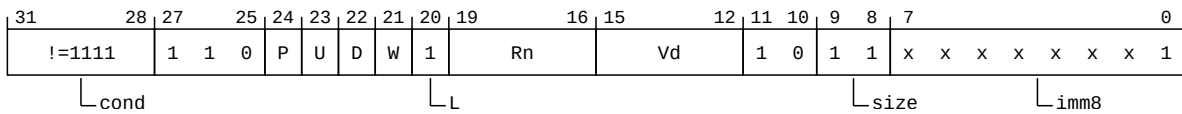
FLDMDBX is the Decrement Before variant of this instruction, and FLDMIAX is the Increment After variant. FLDM*X loads multiple SIMD&FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDL"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{imm8::'00'};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

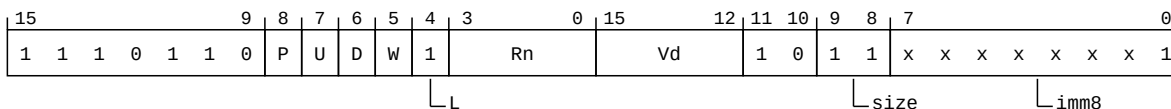
If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If $\text{regs} > 16 \parallel (\text{d+regs}) > 16$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Decrement Before variant

Applies when $(P == 1 \ \&\& \ U == 0 \ \&\& \ W == 1)$

```
FLDMDXB{<c>}{<q>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when $(P == 0 \ \&\& \ U == 1)$

```
FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $\text{regs} == 0$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If $\text{regs} > 16 \parallel (\text{d+regs}) > 16$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: See [Advanced SIMD and floating-point 64-bit move](#) for the T32 instruction set, or [Advanced SIMD and floating-point 64-bit move](#) for the A32 instruction set.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If writeback is not specified, the PC can be used.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

!

Specifies base register writeback, encoded in “W”:

W	!
0	[absent]
1	[present]

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;

    for r = 0 to regs-1 do
        if single_regs then
            S(d+r) = MemA{32}(address);
            address = address+4;
        else
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            address = address+8;

            // Combine the word-aligned words in the correct order for current endianness.
            D(d+r) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
        end;
    end;

    if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

C6.1.6 FSTMDBX, FSTMIAX

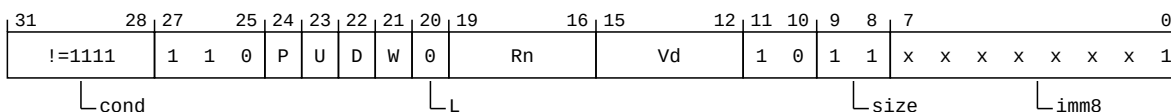
FSTMX stores multiple SIMD&FP registers from the Advanced SIMD and floating-point register file to consecutive locations in using an address from a general-purpose register.

Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
FSTMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

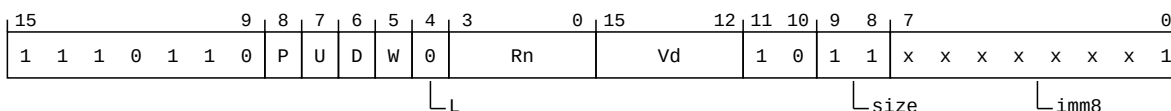
If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If regs > 16 || (d+regs) > 16, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

T1



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
FSTMDBX{<c>}{<q>}<Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
FSTMIAX{<c>} {<q>}    <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $\text{regs} == 0$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If $\text{regs} > 16 \mid\mid (\text{d+regs}) > 16$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: See [Advanced SIMD and floating-point 64-bit move](#) for the T32 instruction set, or [Advanced SIMD and floating-point 64-bit move](#) for the A32 instruction set.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers.

!

Specifies base register writeback, encoded in “W”:

W	!
0	[absent]
1	[present]

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;
    for r = 0 to regs-1 do
        if single_regs then
            MemA{32}(address) = S(d+r);
            address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA{32}(address) = D(d+r)[63:32];
                MemA{32}(address+4) = D(d+r)[31:0];
            else
                MemA{32}(address) = D(d+r)[31:0];
                MemA{32}(address+4) = D(d+r)[63:32];
            end;

            address = address+8;
        end;
    end;

    if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

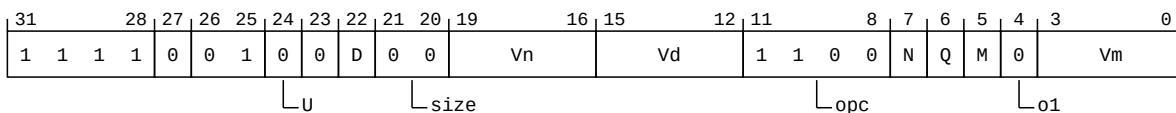
C6.1.7 SHA1C

SHA1 hash update (choose).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA1)



Encoding

SHA1C.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

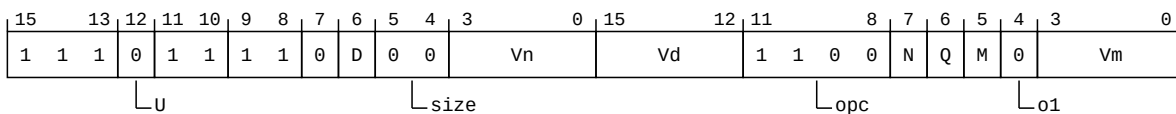
```

if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA1)



Encoding

SHA1C.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var x : bits(128) = Q(d>>1);
    var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
    let w : bits(128) = Q(m>>1);
    for e = 0 to 3 do
        let t : bits(32) = SHAchoose(x[63:32], x[95:64], x[127:96]);
        y = y + ROL(x[31:0], 5) + t + w[e*:32];
        x[63:32] = ROL(x[63:32], 30);
        let yx : bits(160) = ROL(y::x, 32);
        (y, x) = (yx[128+:32], yx[0+:128]);
    end;
    Q(d>>1) = x;
end;

```

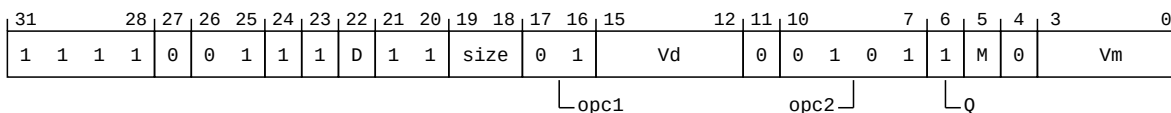
C6.1.8 SHA1H

SHA1 fixed rotate.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA1)



Encoding

SHA1H.32 <Qd>, <Qm>

Decode for this encoding

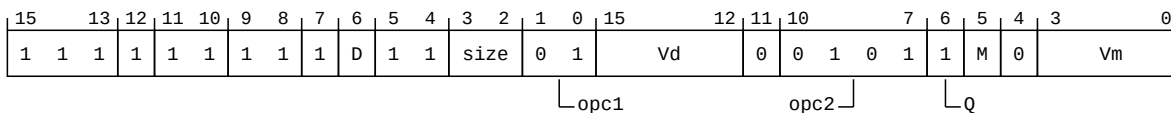
```

if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA1)



Encoding

SHA1H.32 <Qd>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    Q(d>>1) = ZeroExtend{128}(ROL(Q(m>>1)[31:0], 30));
end;
```

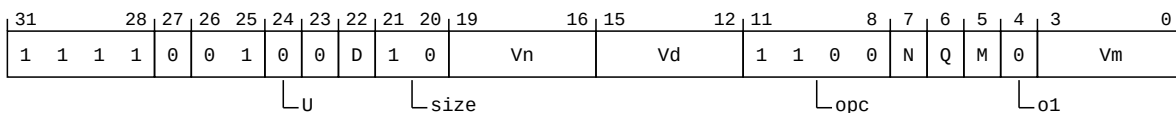
C6.1.9 SHA1M

SHA1 hash update (majority).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA1)



Encoding

SHA1M.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

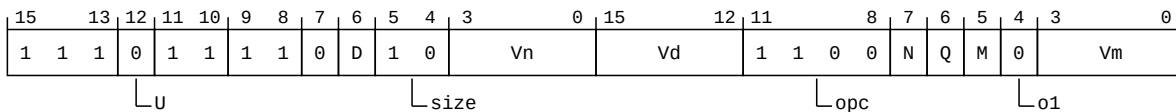
```

if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA1)



Encoding

SHA1M.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var x : bits(128) = Q(d>>1);
    var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
    let w : bits(128) = Q(m>>1);
    for e = 0 to 3 do
        let t : bits(32) = SHAMajority{(x[63:32], x[95:64], x[127:96]);
        y = y + ROL(x[31:0], 5) + t + w[e*:32];
        x[63:32] = ROL(x[63:32], 30);
        let yx : bits(160) = ROL(y::x, 32);
        (y, x) = (yx[128+:32], yx[0+:128]);
    end;
    Q(d>>1) = x;
end;

```

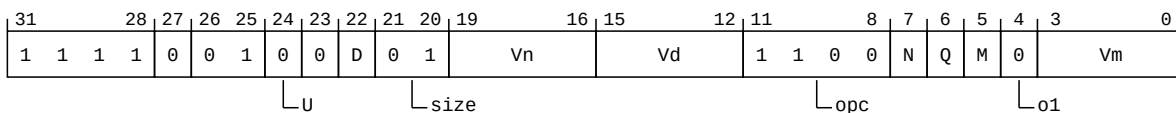
C6.1.10 SHA1P

SHA1 hash update (parity).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA1)



Encoding

SHA1P.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

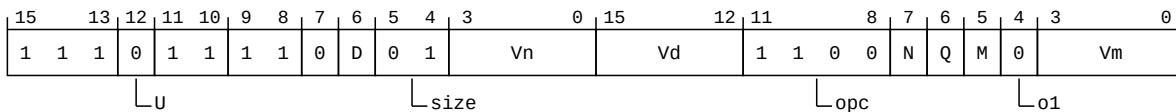
```

if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA1)



Encoding

SHA1P.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var x : bits(128) = Q(d>>1);
    var y : bits(32) = Q(n>>1)[31:0]; // Note: 32 bits wide
    let w : bits(128) = Q(m>>1);
    for e = 0 to 3 do
        let t : bits(32) = SHAparsity(x[63:32], x[95:64], x[127:96]);
        y = y + ROL(x[31:0], 5) + t + w[e*:32];
        x[63:32] = ROL(x[63:32], 30);
        let yx : bits(160) = ROL(y::x, 32);
        (y, x) = (yx[128+:32], yx[0+:128]);
    end;
    Q(d>>1) = x;
end;

```

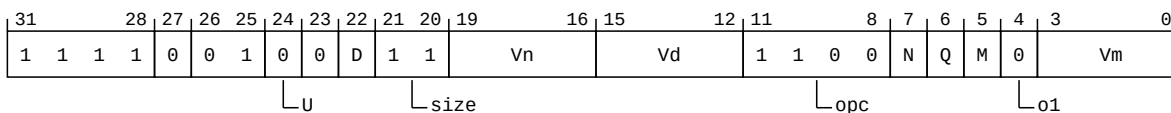
C6.1.11 SHA1SU0

SHA1 schedule update 0.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA1)



Encoding

SHA1SU0.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

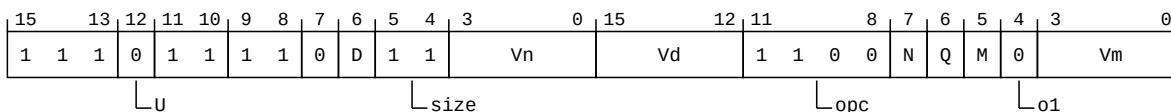
```

if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA1)



Encoding

SHA1SU0.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let op1 : bits(128) = Q(d>>1);
    var op2 : bits(128) = Q(n>>1);
    let op3 : bits(128) = Q(m>>1);
    op2 = op2[63:0] :: op1[127:64];
    Q(d>>1) = op1 XOR op2 XOR op3;
end;
```

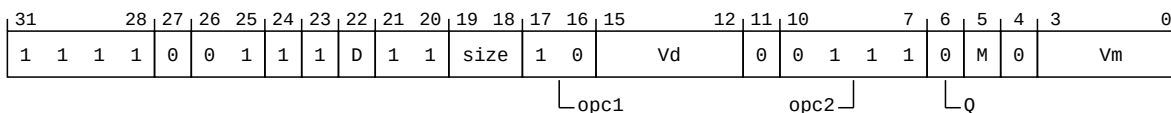
C6.1.12 SHA1SU1

SHA1 schedule update 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA1)



Encoding

SHA1SU1.32 <Qd>, <Qm>

Decode for this encoding

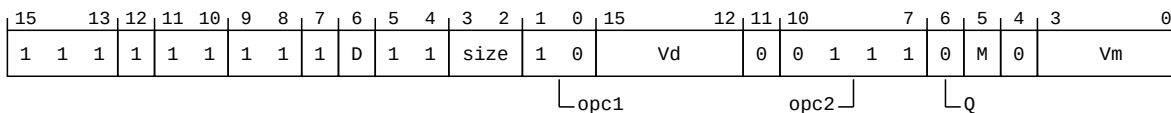
```

if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA1)



Encoding

SHA1SU1.32 <Qd>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA1) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let X : bits(128) = Q(d>>1);
    let Y : bits(128) = Q(m>>1);
    let T : bits(128) = X XOR LSR(Y, 32);
    let W0 : bits(32) = ROL(T[31:0], 1);
    let W1 : bits(32) = ROL(T[63:32], 1);
    let W2 : bits(32) = ROL(T[95:64], 1);
    let W3 : bits(32) = ROL(T[127:96], 1) XOR ROL(T[31:0], 2);
    Q(d>>1) = W3::W2::W1::W0;
end;
```

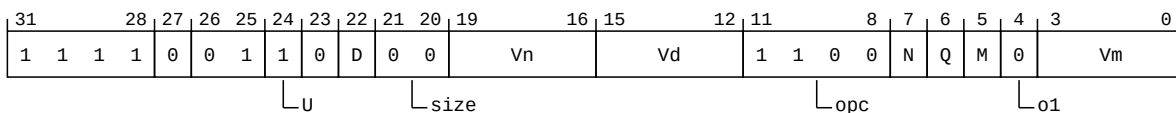
C6.1.13 SHA256H

SHA256 hash update part 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA256)



Encoding

SHA256H.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

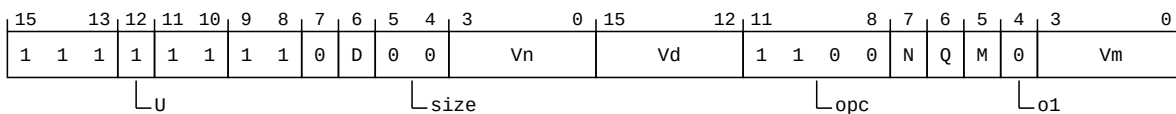
```

if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA256)



Encoding

SHA256H.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let X : bits(128) = Q(d>>1);
    let Y : bits(128) = Q(n>>1);
    let W : bits(128) = Q(m>>1);
    let part1 : boolean = TRUE;
    Q(d>>1) = SHA256hash(X, Y, W, part1);
end;
```

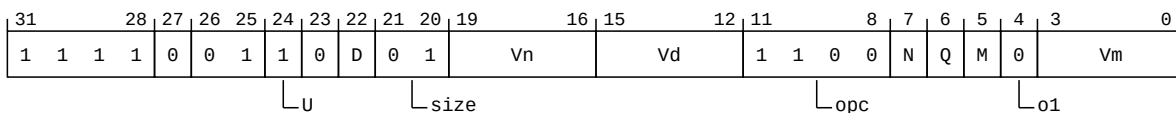
C6.1.14 SHA256H2

SHA256 hash update part 2.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA256)



Encoding

SHA256H2.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

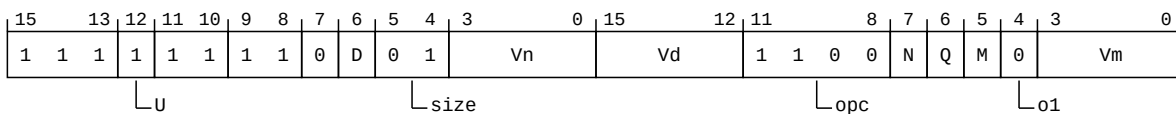
```

if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA256)



Encoding

SHA256H2.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let X : bits(128) = Q(n>>1);
    let Y : bits(128) = Q(d>>1);
    let W : bits(128) = Q(m>>1);
    let part1 : boolean = FALSE;
    Q(d>>1) = SHA256hash(X, Y, W, part1);
end;
```

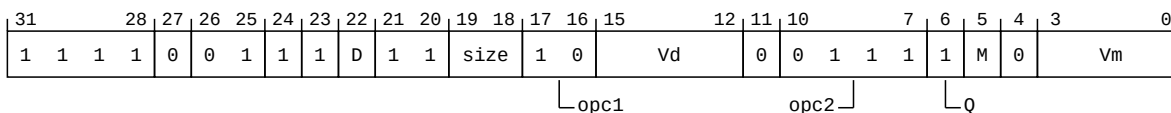
C6.1.15 SHA256SU0

SHA256 schedule update 0.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA256)



Encoding

SHA256SU0.32 <Qd>, <Qm>

Decode for this encoding

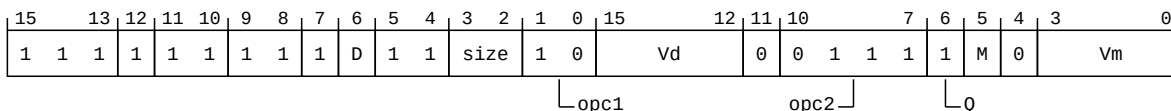
```

if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA256)



Encoding

SHA256SU0.32 <Qd>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if size != '10' then Undefined(); end;
if Vd[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    var result : bits(128);
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    let x : bits(128) = Q(d>>1);
    let y : bits(128) = Q(m>>1);
    let t : bits(128) = y[31:0] :: x[127:32];
    for e = 0 to 3 do
        var elt : bits(32) = t[e*:32];
        elt = ROR(elt, 7) XOR ROR(elt, 18) XOR LSR(elt, 3);
        result[e*:32] = elt + x[e*:32];
    end;
    Q(d>>1) = result;
end;
```

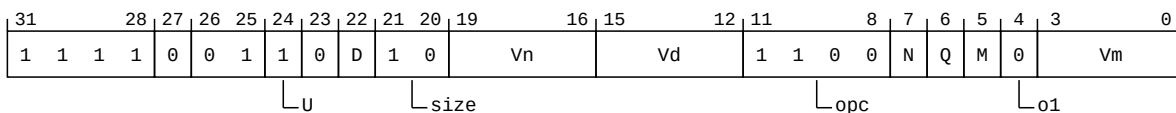
C6.1.16 SHA256SU1

SHA256 schedule update 1.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_SHA256)



Encoding

SHA256SU1.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

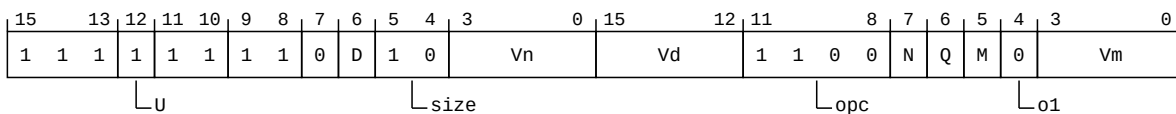
```

if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1

(FEAT_SHA256)



Encoding

SHA256SU1.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if !IsFeatureImplemented(FEAT_SHA256) then Undefined(); end;
if Q != '1' then Undefined(); end;
if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckCryptoEnabled32();
    var elt : bits(32);
    var result : bits(128);
    let x : bits(128) = Q(d>>1);
    let y : bits(128) = Q(n>>1);
    let z : bits(128) = Q(m>>1);
    let T0 : bits(128) = z[31:0] :: y[127:32];

    var T1 : bits(64) = z[127:64];
    for e = 0 to 1 do
        elt = T1[e*:32];
        elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
        elt = elt + x[e*:32] + T0[e*:32];
        result[e*:32] = elt;
    end;

    T1 = result[63:0];
    for e = 2 to 3 do
        elt = T1[(e - 2)*:32];
        elt = ROR(elt, 17) XOR ROR(elt, 19) XOR LSR(elt, 10);
        elt = elt + x[e*:32] + T0[e*:32];
        result[e*:32] = elt;
    end;

    Q(d>>1) = result;
end;

```

C6.1.17 VABA

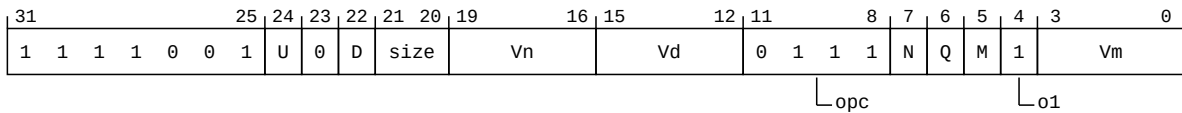
Vector Absolute Difference and Accumulate subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

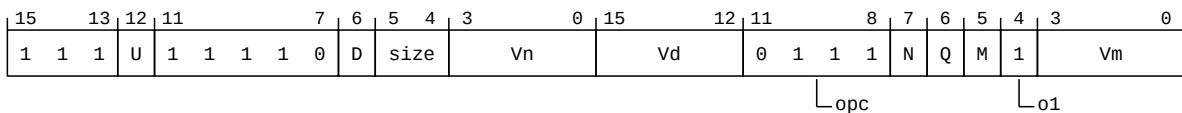
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + absdiff;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + absdiff;
            end;
        end;
    end;
end;
```

C6.1.18 VABAL

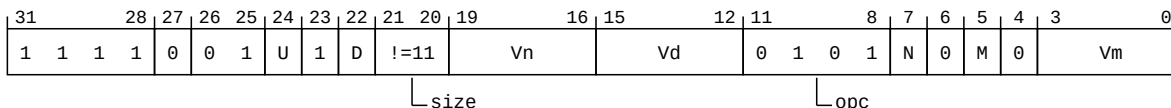
Vector Absolute Difference and Accumulate Long subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

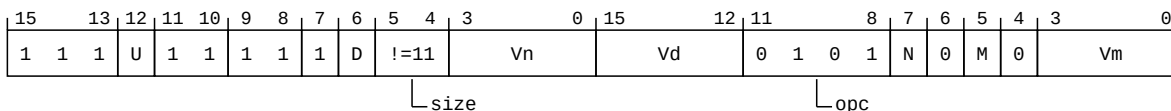
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;

```

T1



Encoding

VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for r = 0 to regs-1 do
    for e = 0 to elements-1 do
```

```
    let op1elt : bits(esize) = Din(n+r)[e*:esize];
    let op2elt : bits(esize) = Din(m+r)[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let absdiff : integer = Abs(element1 - element2);
    if long_destination then
        Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + absdiff;
    else
        D(d+r)[e*:esize] = Din(d+r)[e*:esize] + absdiff;
    end;
end;
end;
end;
```

C6.1.19 VABD (floating-point)

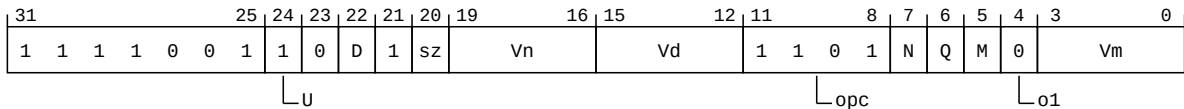
Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are floating-point numbers of the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

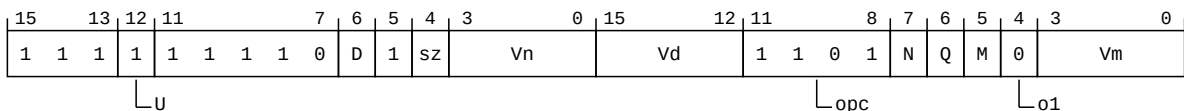
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let esize : integer() = 32;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let esize : integer{} = 32;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            D(d+r)[e*:esize] = FPAbs{esize}(FPSub{esize}(op1elt, op2elt, fpcr), fpcr);
        end;
    end;
end;
```

C6.1.20 VABD (integer)

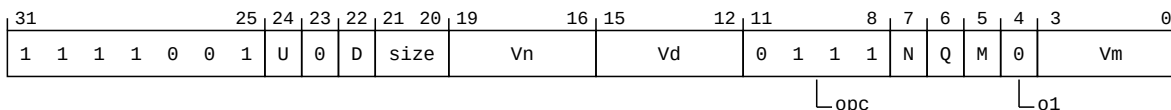
Vector Absolute Difference (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

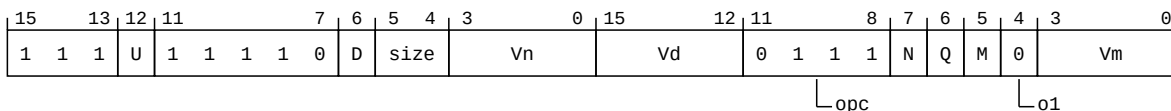
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let absdiff : integer = Abs(element1 - element2);
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = absdiff[2*esize-1:0];
            else
                D(d+r)[e*:esize] = absdiff[esize-1:0];
            end;
        end;
    end;
end;
```

C6.1.21 VABDL (integer)

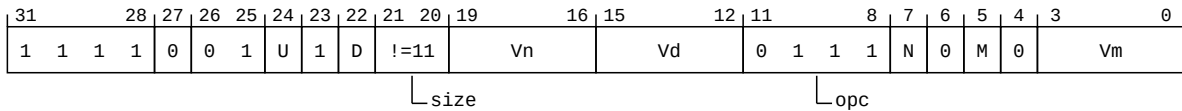
Vector Absolute Difference Long (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

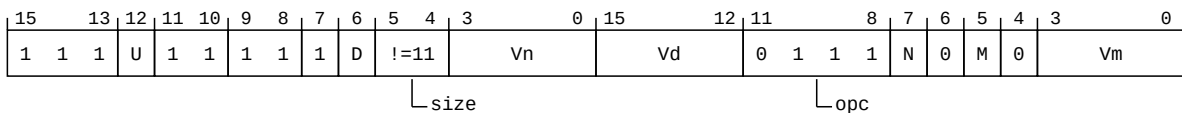
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;

```

T1



Encoding

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
```

```
    let op1elt : bits(esize) = Din(n+r)[e*:esize];
    let op2elt : bits(esize) = Din(m+r)[e*:esize];
    let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    let absdiff : integer = Abs(element1 - element2);
    if long_destination then
        Q(d>>1)[e*:(2*esize)] = absdiff[2*esize-1:0];
    else
        D(d+r)[e*:esize] = absdiff[esize-1:0];
    end;
end;
end;
end;
```

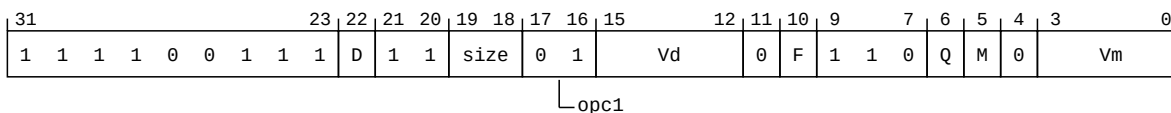
C6.1.22 VABS

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VABS{<c>}{<q>}.<dt> <Qd>, <Qm>

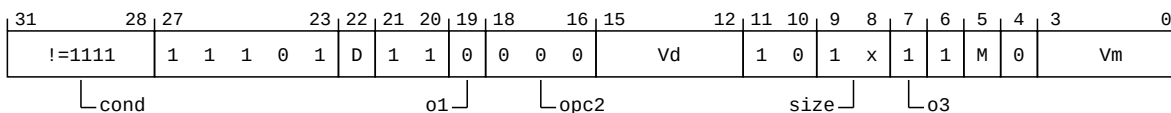
Decode for all variants of this encoding

```

if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '1l' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '1l' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

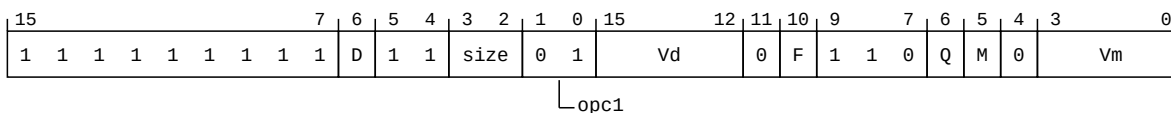
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VABS{<c>}{<q>},<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VABS}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \langle Qd \rangle, \quad \langle Qm \rangle$$

Decode for all variants of this encoding

```

if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

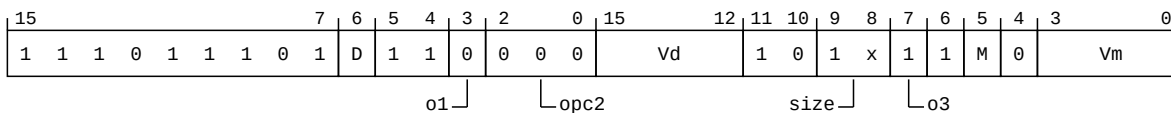
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                if floating_point then
                    D(d+r)[e*:esize] = FPAbs{esize}(D(m+r)[e*:esize], fpcr);
                else
                    let result : integer = Abs(SInt(D(m+r)[e*:esize]));
                    D(d+r)[e*:esize] = result[esize-1:0];
                end;
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
```



```
        when 32 => S(d) = FPAbs{32}(S(m), fpcr);  
        when 64 => D(d) = FPAbs{64}(D(m), fpcr);  
    end;  
end;  
end;
```

C6.1.23 VACGE

Vector Absolute Compare Greater Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

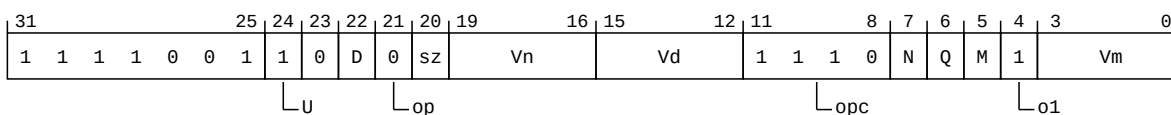
The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VACLE](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

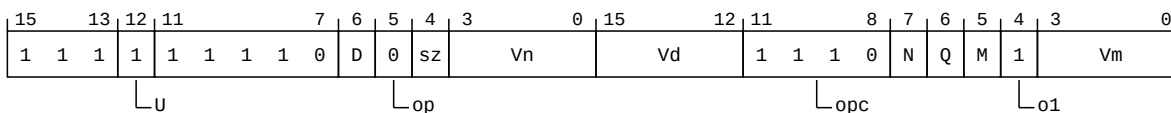
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = FPAbs{}(D(n+r)[e*:esize], fpcr);
            let op2 : bits(esize) = FPAbs{}(D(m+r)[e*:esize], fpcr);
            var test_passed : boolean;
            if or_equal then
                test_passed = FPCompareGE{esize}(op1, op2, fpcr);
            else
                test_passed = FPCompareGT{esize}(op1, op2, fpcr);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;

```

C6.1.24 VACGT

Vector Absolute Compare Greater Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

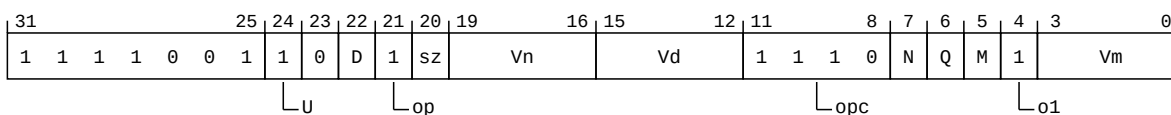
The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VACLT](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

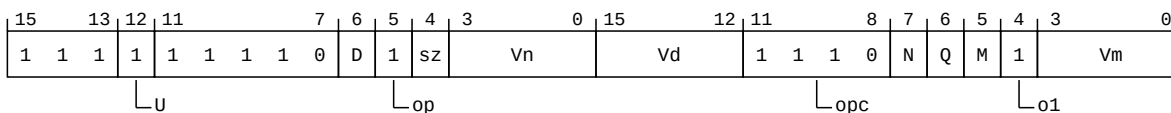
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let or_equal : boolean = (op == '0');
let esize : integer{} = 32;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = FPAbs{}(D(n+r)[e*:esize], fpcr);
            let op2 : bits(esize) = FPAbs{}(D(m+r)[e*:esize], fpcr);
            var test_passed : boolean;
            if or_equal then
                test_passed = FPCompareGE{esize}(op1, op2, fpcr);
            else
                test_passed = FPCompareGT{esize}(op1, op2, fpcr);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;

```

C6.1.25 VACLE

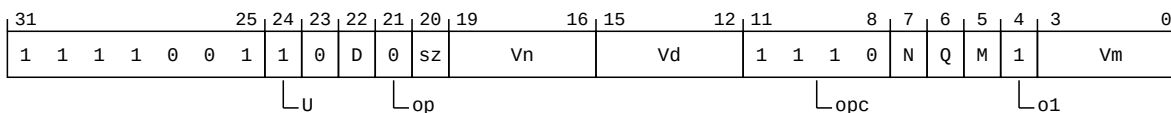
Vector Absolute Compare Less Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of [VACGE](#). This means:

- The encodings in this description are named to match the encodings of [VACGE](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VACGE](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

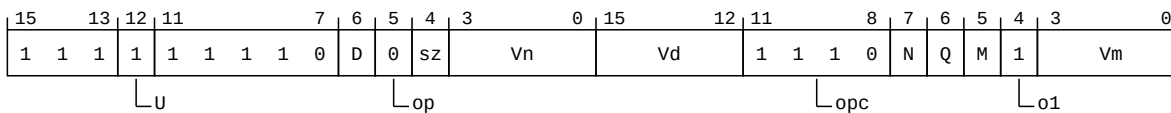
Applies when (Q == 1)

VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VACGE](#) gives the operational pseudocode for this instruction.

C6.1.26 VACLT

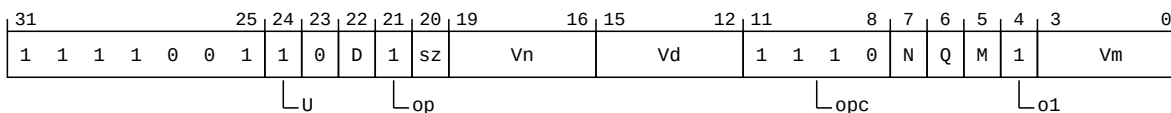
Vector Absolute Compare Less Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of [VACGT](#). This means:

- The encodings in this description are named to match the encodings of [VACGT](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VACGT](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

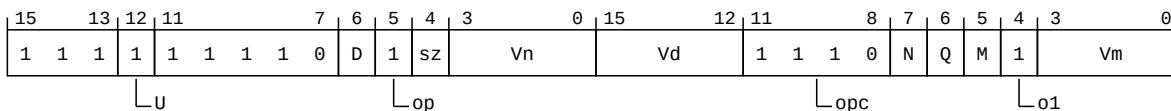
Applies when (Q == 1)

VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VACGT](#) gives the operational pseudocode for this instruction.

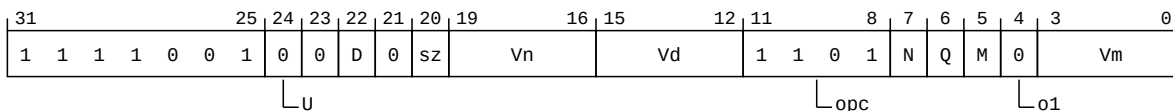
C6.1.27 VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPXEC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VADD}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \{\langle Qd \rangle, \} \langle Qn \rangle, \langle Qm \rangle$$

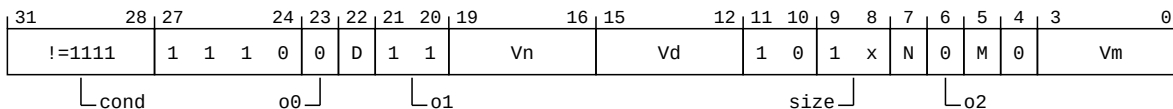
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VADD{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VADD{<c>} {<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

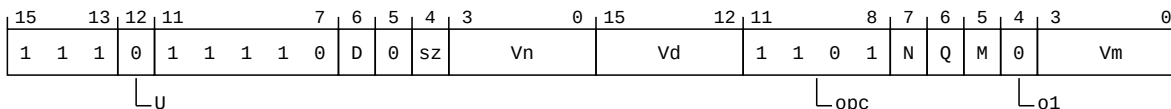
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VADD}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \{\langle Qd \rangle, \} \langle Qn \rangle, \langle Qm \rangle$$

Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

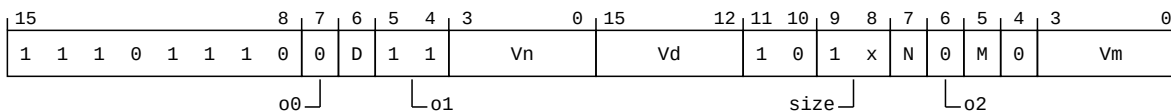
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

VADD{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

VADD{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*:esize] = FPAdd{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
            end;
        end;
    else // VFP instruction
```

```
    let fpcr : FPCR_Type = EffectiveFPCR();  
    case esize of  
    when 32 =>  
        S(d) = FPAdd{32}(S(n), S(m), fpcr);  
    when 64 =>  
        D(d) = FPAdd{64}(D(n), D(m), fpcr);  
    end;  
end;  
end;
```

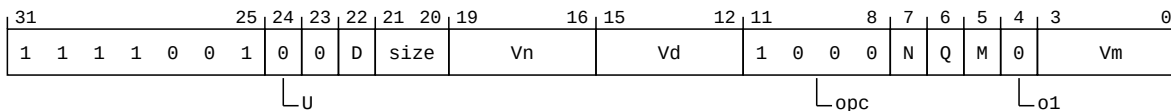
C6.1.28 VADD (integer)

Vector Add (integer) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPT](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

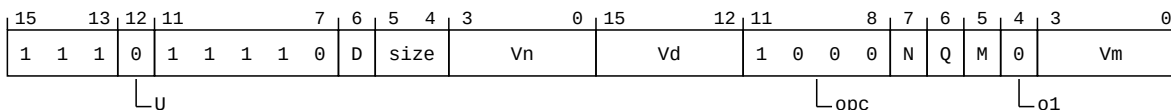
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = D(n+r)[e*:esize] + D(m+r)[e*:esize];
        end;
    end;
end;
```

C6.1.29 VADDHN

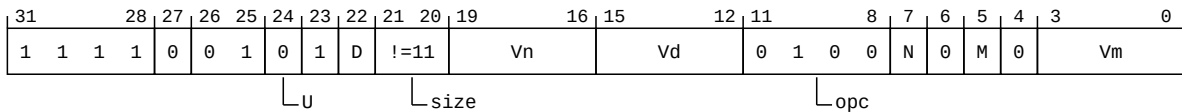
Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. For rounded results, see [VRADDHN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

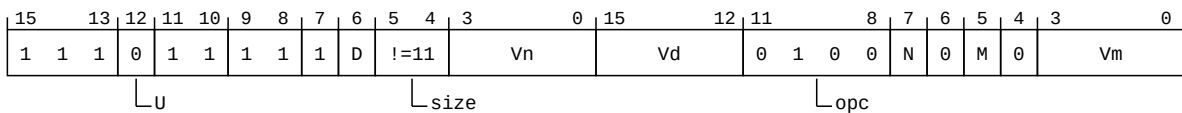
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

T1



Encoding

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let result : bits(2*esize) = Qin(n>>1)[e:(2*esize)] + Qin(m>>1)[e:(2*esize)];
        D(d)[e*:esize] = result[2*esize-1:esize];
    end;
end;
```

C6.1.30 VADDL

Vector Add Long adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	9	8	7	6	5	4	3	0		
1	1	1	1	0	0	1	U	1	D	!=11	Vn		Vd		0	0	0	0	N	0	M	0	Vm	
size											op													

Encoding

VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

T1

15	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	9	8	7	6	5	4	3	0		
1	1	1	U	1	1	1	1	1	D	!=11	Vn		Vd			0	0	0	0	N	0	M	0	Vm	
size											op														

Encoding

VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the second operand vector, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vaddw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:(esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:(esize)];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 + element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;

```

C6.1.31 VADDW

Vector Add Wide adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	9	8	7	6	5	4	3	0		
1	1	1	1	0	0	1	U	1	D	!=11	Vn		Vd		0	0	0	1	N	0	M	0	Vm	
size											op													

Encoding

VADDW{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

T1

15	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	9	8	7	6	5	4	3	0	
1	1	1	U	1	1	1	1	1	D	!=11	Vn		Vd		0	0	0	1	N	0	M	0	Vm	
size												op												

Encoding

VADDW{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vaddw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the second operand vector, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vaddw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 + element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

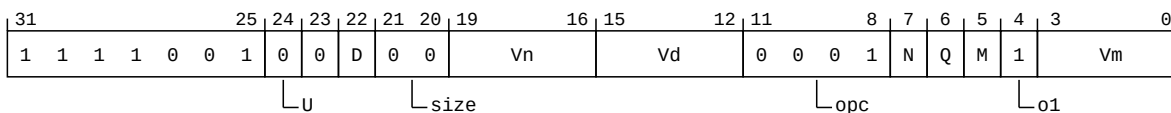
C6.1.32 VAND (register)

Vector Bitwise AND (register) performs a bitwise AND operation between two registers, and places the result in the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VAND{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

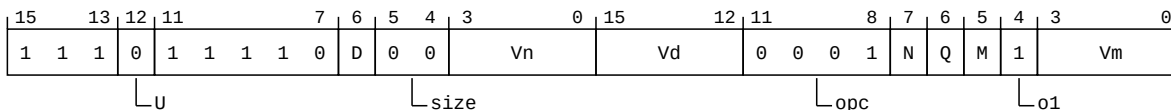
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VAND{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) AND D(m+r);
    end;
end;
```

C6.1.33 VAND (immediate)

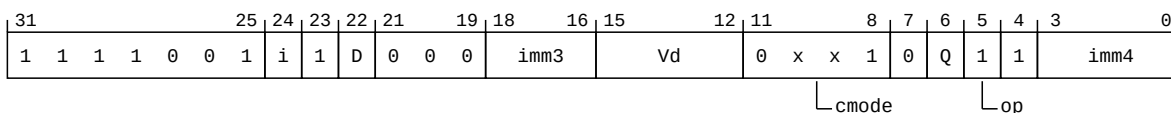
Vector Bitwise AND (immediate) performs a bitwise AND between a register value and an immediate value, and returns the result into the destination vector.

This is a pseudo-instruction of [VBIC \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [VBIC \(immediate\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VBIC \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

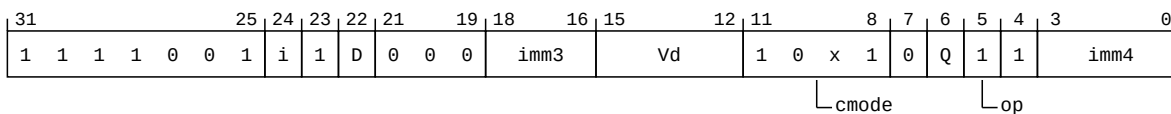
Applies when (Q == 1)

VAND{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

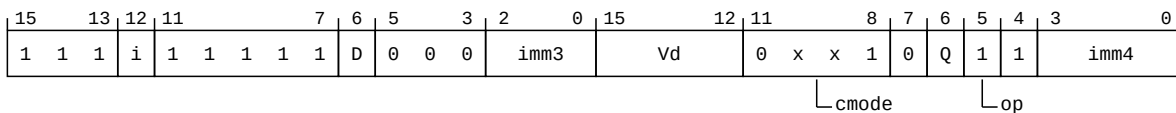
Applies when (Q == 1)

VAND{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

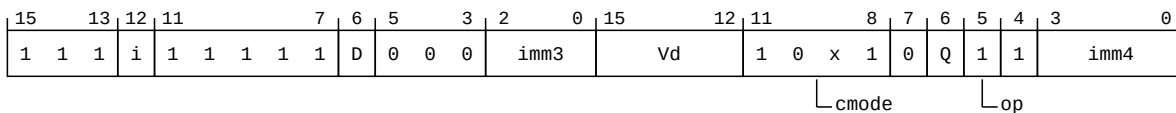
Applies when (Q == 1)

VAND{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VAND{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VAND{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Operation

The description of [VBIC \(immediate\)](#) gives the operational pseudocode for this instruction.

C6.1.34 VBIC (immediate)

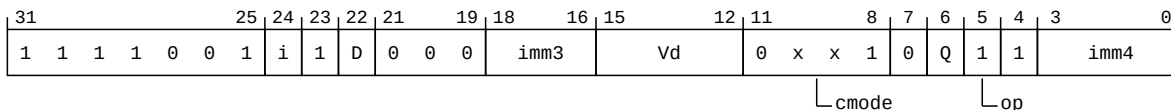
Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VAND \(immediate\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VBIC{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

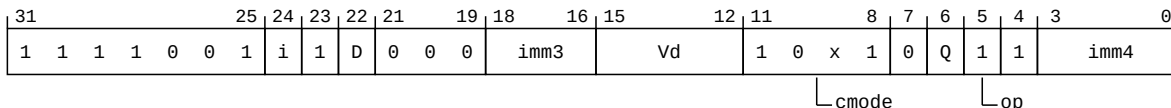
Applies when (Q == 1)

```
VBIC{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VBIC{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

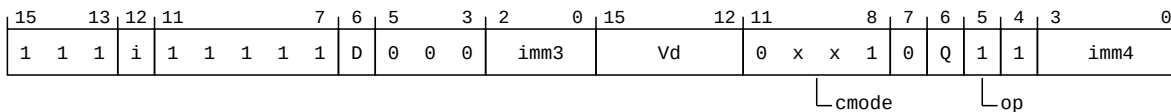
Applies when (Q == 1)

```
VBIC{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDEExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VBIC{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

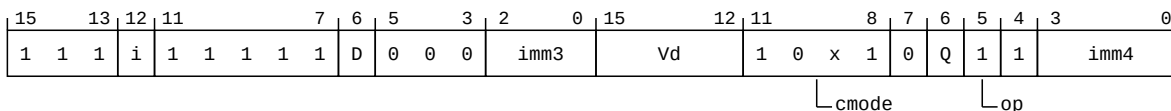
Applies when (Q == 1)

```
VBIC{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDEExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VBIC{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VBIC{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>
```


Decode for all variants of this encoding

```

if cmode[0] == '0' || cmode[3:2] == '11' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;

```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(d+r) AND NOT(imm64);
    end;
end;

```

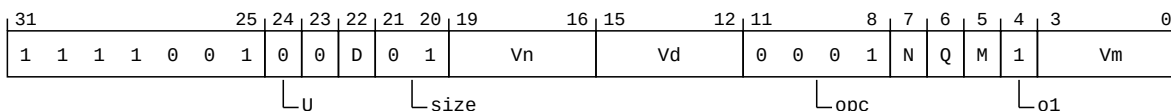
C6.1.35 VBIC (register)

Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIC{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIC{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

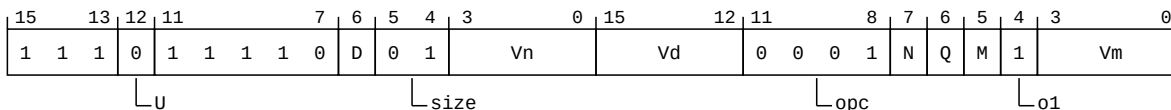
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIC{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIC{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) AND NOT(D(m+r));
    end;
end;
```

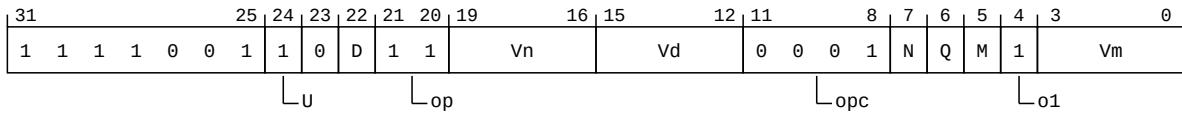
C6.1.36 VBIF

Vector Bitwise Insert if False inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIF{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIF{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

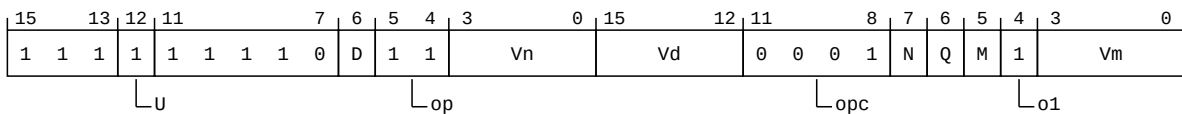
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VBIF{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBITF{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIT; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps_VBIF => D(d+r) = (D(d+r) AND D(m+r)) OR (D(n+r) AND NOT(D(m+r)));
            when VBitOps_VBIT => D(d+r) = (D(n+r) AND D(m+r)) OR (D(d+r) AND NOT(D(m+r)));
            when VBitOps_VBSL => D(d+r) = (D(n+r) AND D(d+r)) OR (D(m+r) AND NOT(D(d+r)));
        end;
    end;
end;
```

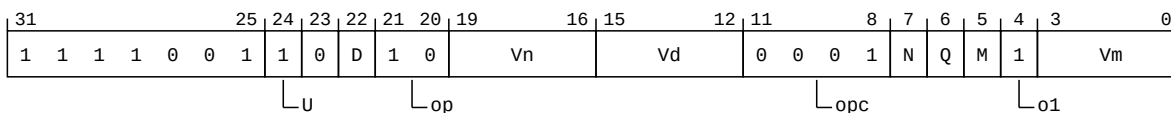
C6.1.37 VBIT

Vector Bitwise Insert if True inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VBIT{<c>}{<q>}{.<dt>}{<Dd>,<Dn>,<Dm>}
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VBIT{<c>}{<q>}{.<dt>}{<Od>,<On>,<Om>}
```

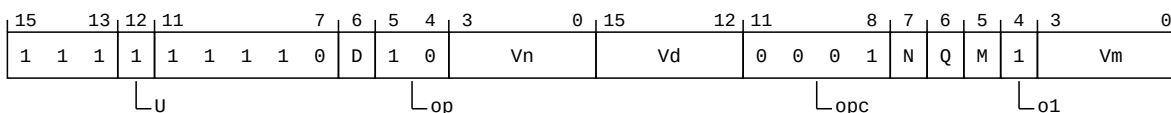
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VBIT{<c>}{<q>}{,<dt>}{<Dd>,<Dn>,<Dm>}
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBIT{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps_VBIF => D(d+r) = (D(d+r) AND D(m+r)) OR (D(n+r) AND NOT(D(m+r)));
            when VBitOps_VBIT => D(d+r) = (D(n+r) AND D(m+r)) OR (D(d+r) AND NOT(D(m+r)));
            when VBitOps_VBSL => D(d+r) = (D(n+r) AND D(d+r)) OR (D(m+r) AND NOT(D(d+r)));
        end;
    end;
end;
end;

```

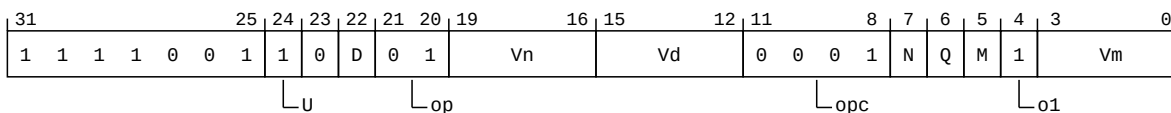
C6.1.38 **VBSL**

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VBSL{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VBSL{<c>}{<q>}{.<dt>}{<Od>,<On>,<Om>}

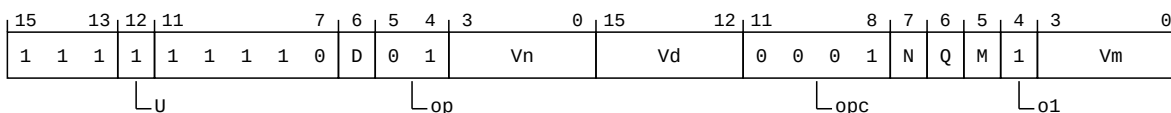
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VBSL{<c>}{<q>}{.<dt>}{<Dd>,<Dn>,<Dm>}

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VBSL{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if op == '00' then See("VEOR"); end;
var operation : VBitOps;
if op == '01' then operation = VBitOps_VBSL; end;
if op == '10' then operation = VBitOps_VBIT; end;
if op == '11' then operation = VBitOps_VBIF; end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        case operation of
            when VBitOps_VBIF => D(d+r) = (D(d+r) AND D(m+r)) OR (D(n+r) AND NOT(D(m+r)));
            when VBitOps_VBIT => D(d+r) = (D(n+r) AND D(m+r)) OR (D(d+r) AND NOT(D(m+r)));
            when VBitOps_VBSL => D(d+r) = (D(n+r) AND D(d+r)) OR (D(m+r) AND NOT(D(d+r)));
        end;
    end;
end;
end;

```

C6.1.39 VCEQ (immediate #0)

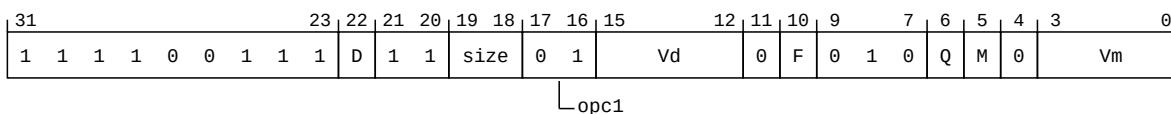
Vector Compare Equal to Zero takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

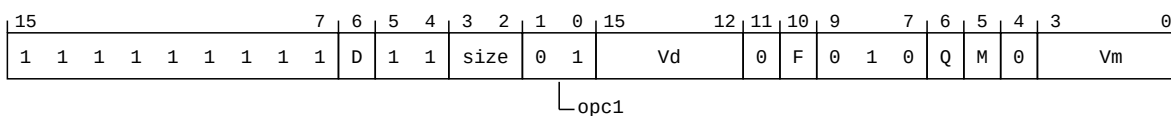
Decode for all variants of this encoding

```

if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCEO{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero({'0'});
                test_passed = FPCompareEQ(esize)(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (D(m+r)[e*:esize] == Zeros(esize));
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;
```

C6.1.40 VCEQ (register)

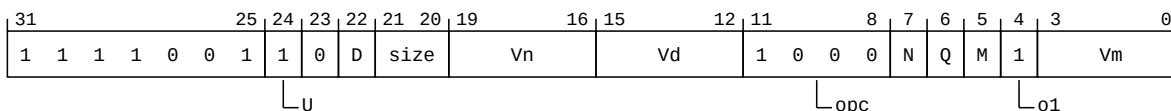
Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

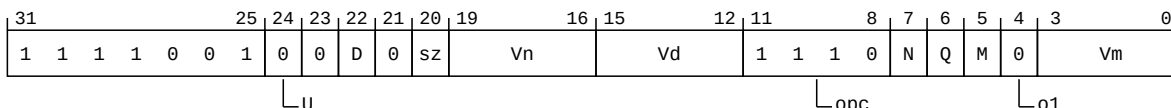
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let int_operation : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

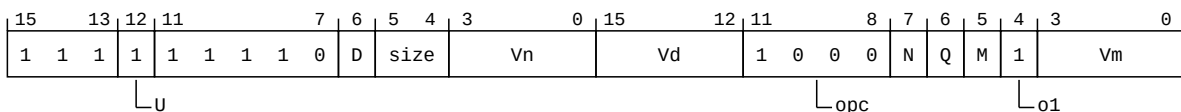
Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, <Qn>, <Qm>}

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let int_operation : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, <Dn>, <Dm>}

Encoding for the 128-bit SIMD vector variant

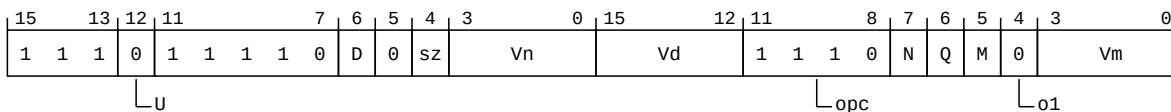
Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, <Qn>, <Qm>}

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let int_operation : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let int_operation : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            var test_passed : boolean;
            if int_operation then
                test_passed = (op1elt == op2elt);
            else
                test_passed = FPCompareEQ{esize}(op1elt, op2elt, StandardFPCR());
            end;
            D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};
        end;
    end;
end;
end;

```

C6.1.41 VCGE (immediate #0)

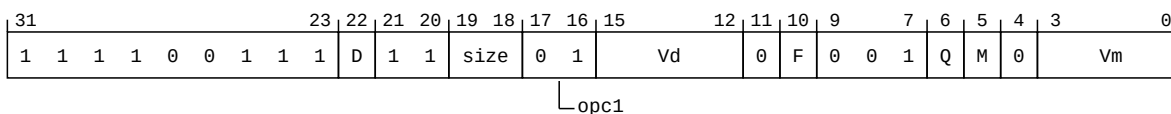
Vector Compare Greater Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

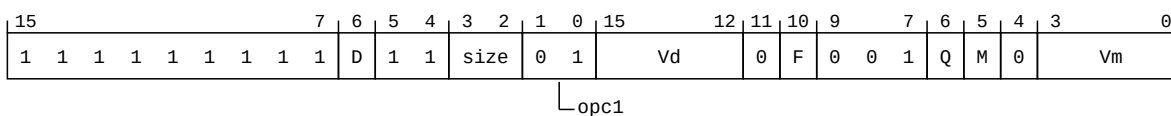
Decode for all variants of this encoding

```

if size == '1l' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero({'0'});
                test_passed = FPCompareGE(esize)(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) >= 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;
```

C6.1.42 VCGE (register)

Vector Compare Greater Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

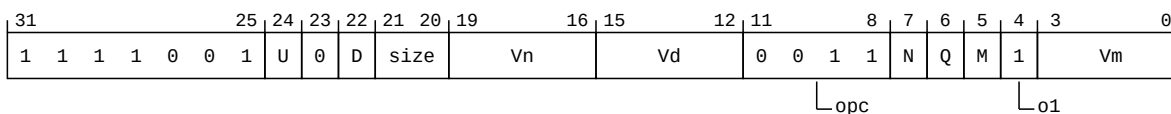
The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VCLE \(register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

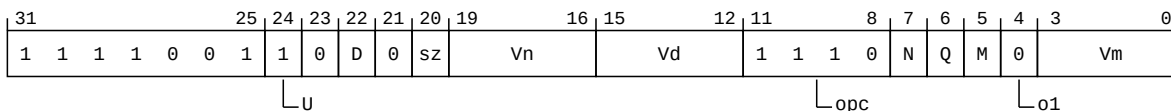
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGEType = if U == '1' then VCGEType_unsigned else VCGEType_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

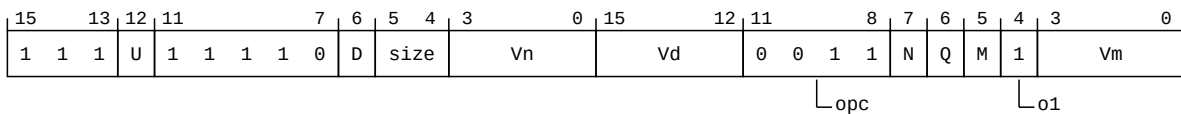
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let vtype : VCGType = VCGType_fp;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

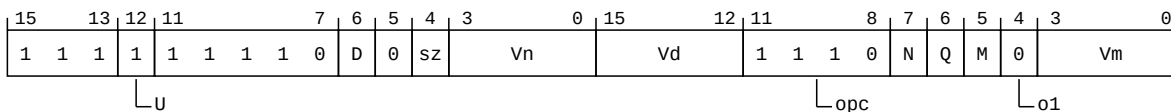
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGType = if U == '1' then VCGType_unsigned else VCGType_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let vtype : VCGEType = VCGEType_fp;
let esize : integer{} = 32;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
    
```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            var test_passed : boolean;
            case vtype of
                when VCGType_signed => test_passed = (SInt(op1elt) >= SInt(op2elt));
                when VCGType_unsigned => test_passed = (UInt(op1elt) >= UInt(op2elt));
                when VCGType_fp => test_passed = FPCompareGE{esize}(op1elt, op2elt,
                                                                    StandardFPCR());
            end;
        end;
    end;

```

```
        D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};  
    end;  
end;  
end;
```

C6.1.43 VCGT (immediate #0)

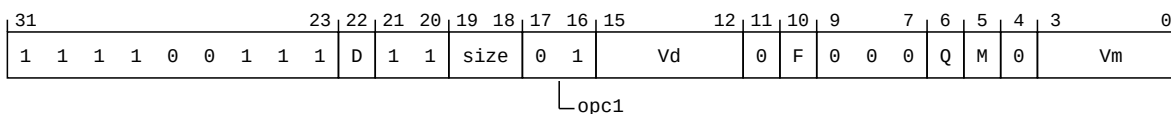
Vector Compare Greater Than Zero takes each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

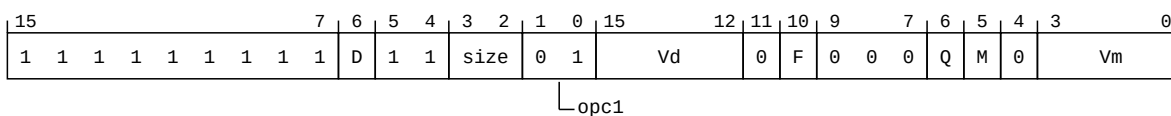
Decode for all variants of this encoding

```

if size == '1l' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{}('0');
                test_passed = FPCompareGT(esize)(D(m+r)[e*:esize], zero, StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) > 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;
```

C6.1.44 VCGT (register)

Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

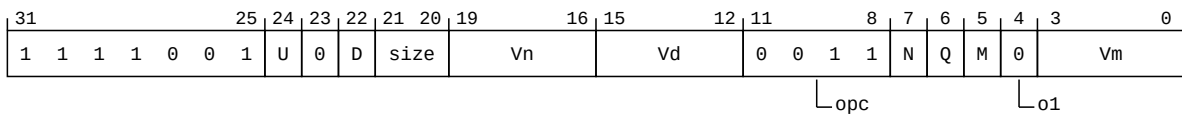
The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VCLT \(register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

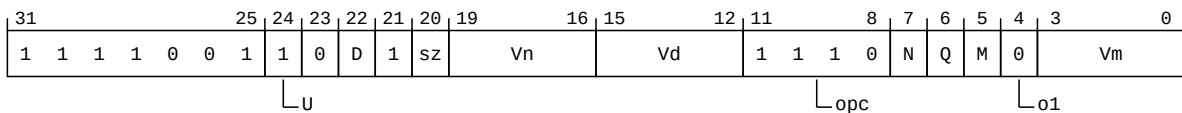
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGTtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

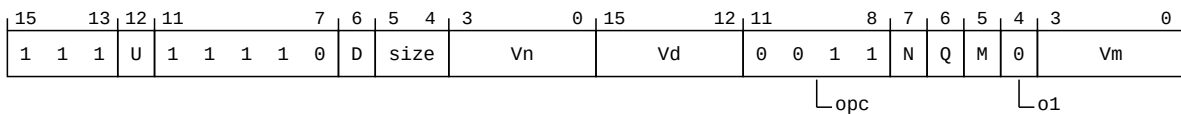
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let vtype : VCGTtype = VCGTtype_fp;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

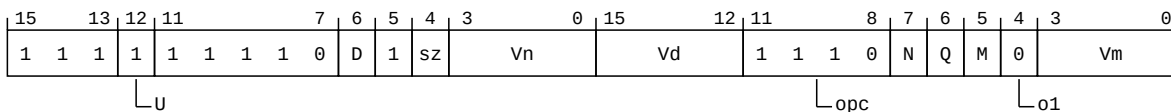
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let vtype : VCGTtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, <Dn>, <Dm>}

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, <Qn>, <Qm>}

Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let vtype : VCGTtype = VCGTtype_fp;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            var test_passed : boolean;
            case vtype of
                when VCGTtype_signed => test_passed = (SInt(op1elt) > SInt(op2elt));
                when VCGTtype_unsigned => test_passed = (UInt(op1elt) > UInt(op2elt));
                when VCGTtype_fp => test_passed = FPCompareGT{esize}(op1elt, op2elt,
                                                                    StandardFPCR());
            end;
        end;
    end;

```

```
        D(d+r)[e*:esize] = if test_passed then Ones{esize} else Zeros{esize};  
    end;  
end;  
end;
```

C6.1.45 VCLE (immediate #0)

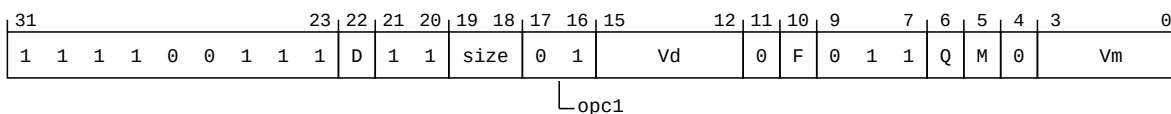
Vector Compare Less Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

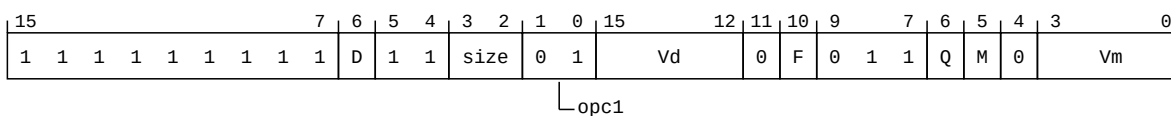
Decode for all variants of this encoding

```

if size == '1l' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero({'0'});
                test_passed = FPCompareGE(esize)(zero, D(m+r)[e*:esize], StandardFPCR());
            else
                test_passed = (SInt(D(m+r)[e*:esize]) <= 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;
```

C6.1.46 VCLE (register)

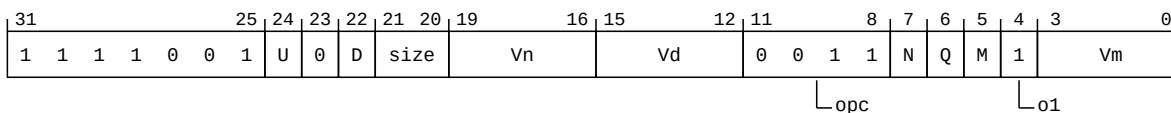
Vector Compare Less Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of [VCGE \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VCGE \(register\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VCGE \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

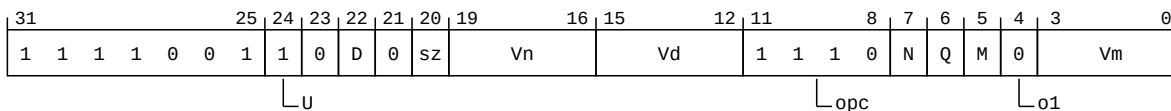
Applies when (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

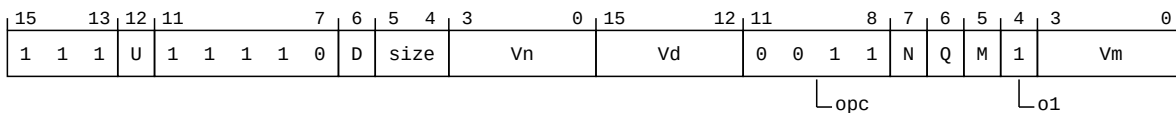
Applies when ($Q == 1$)

$VCLE\{<c>\}\{<q>\}.<dt>\{<Qd>, <Qn>, <Qm>\}$

is equivalent to

$VCGE\{<c>\}\{<q>\}.<dt>\{<Qd>, <Qm>, <Qn>\}$

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

$VCLE\{<c>\}\{<q>\}.<dt>\{<Dd>, <Dn>, <Dm>\}$

is equivalent to

$VCGE\{<c>\}\{<q>\}.<dt>\{<Dd>, <Dm>, <Dn>\}$

Encoding for the 128-bit SIMD vector variant

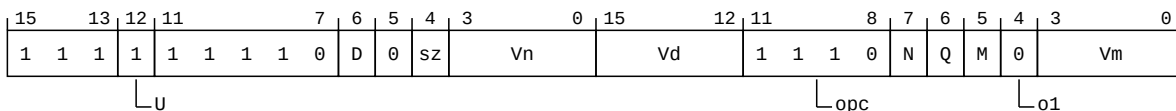
Applies when ($Q == 1$)

$VCLE\{<c>\}\{<q>\}.<dt>\{<Qd>, <Qn>, <Qm>\}$

is equivalent to

$VCGE\{<c>\}\{<q>\}.<dt>\{<Qd>, <Qm>, <Qn>\}$

T2



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

$VCLE\{<c>\}\{<q>\}.<dt>\{<Dd>, <Dn>, <Dm>\}$

is equivalent to

$VCGE\{<c>\}\{<q>\}.<dt>\{<Dd>, <Dm>, <Dn>\}$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VCGE \(register\)](#) gives the operational pseudocode for this instruction.

C6.1.47 VCLS

Vector Count Leading Sign Bits determines the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The result does not include the topmost bit itself.

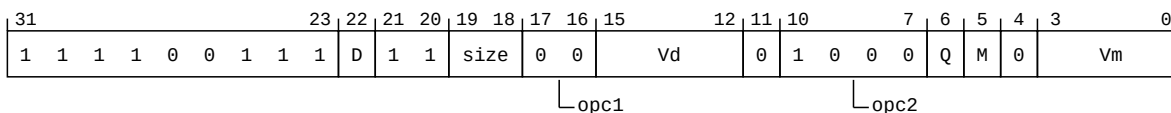
The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>

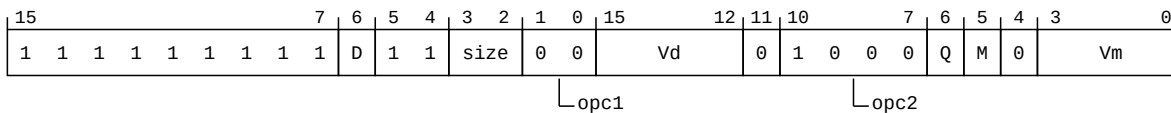
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = CountLeadingSignBits(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;
```

C6.1.48 VCLT (immediate #0)

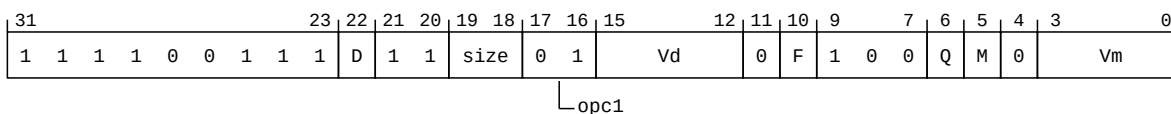
Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0
```

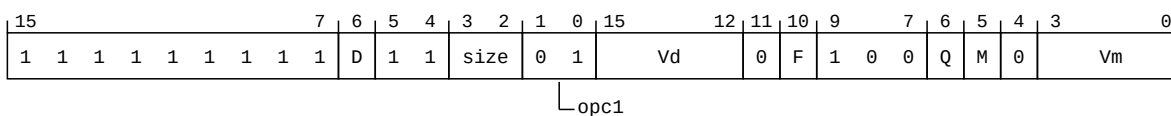
Decode for all variants of this encoding

```

if size == '1l' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dm>, #0
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qm>, #0

Decode for all variants of this encoding

```
if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            var test_passed : boolean;
            if floating_point then
                let zero : bits(esize) = FPZero{}('0');
                test_passed = FPCompareGT(esize){zero, D(m+r)[e*:esize], StandardFPCR()};
            else
                test_passed = (SInt(D(m+r)[e*:esize]) < 0);
            end;
            D(d+r)[e*:esize] = if test_passed then Ones(esize) else Zeros(esize);
        end;
    end;
end;
```

C6.1.49 VCLT (register)

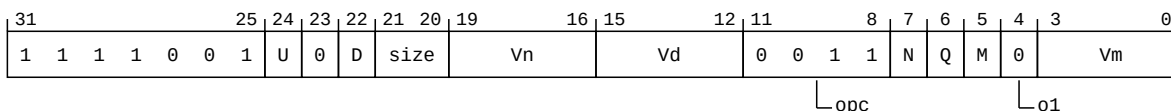
Vector Compare Less Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of [VCGT \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VCGT \(register\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VCGT \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

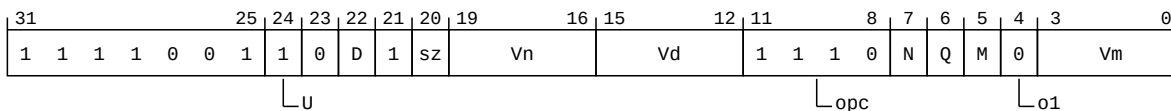
Applies when (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

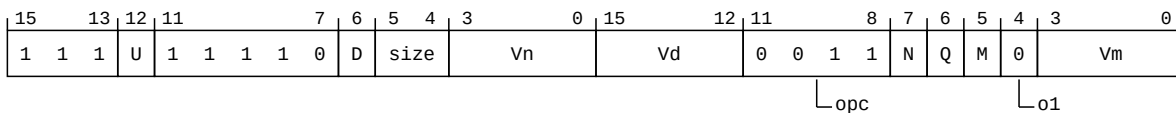
Applies when ($Q == 1$)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

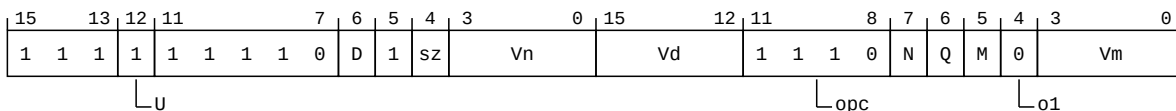
Applies when ($Q == 1$)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

T2



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “T1 128-bit SIMD vector”, and “T1 64-bit SIMD vector” variants: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For the “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VCGT \(register\)](#) gives the operational pseudocode for this instruction.

C6.1.50 VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

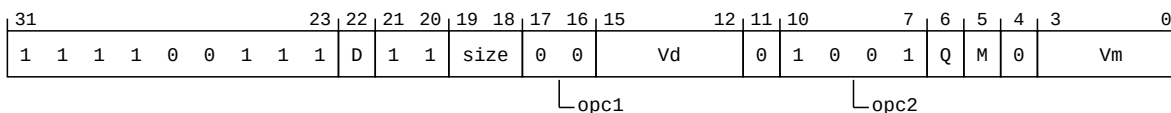
The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>

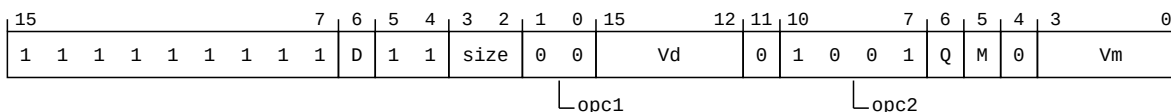
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = CountLeadingZeroBits(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;
```

C6.1.51 VCMF

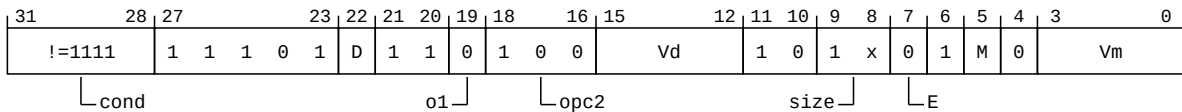
Vector Compare compares two floating-point registers, or one floating-point register and zero. It writes the result to the **FPSCR** flags. These are normally transferred to the **PSTATE**.{N, Z, C, V} Condition flags by a subsequent **VMRS** instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

Depending on settings in the **CPACR**, **NSACR**, and **HCPTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (**A1** and **A2**) and T32 (**T1** and **T2**).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

`VCMF{<c>}{<q>}.F32 <Sd>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when (size == 11)

`VCMF{<c>}{<q>}.F64 <Dd>, <Dm>`

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

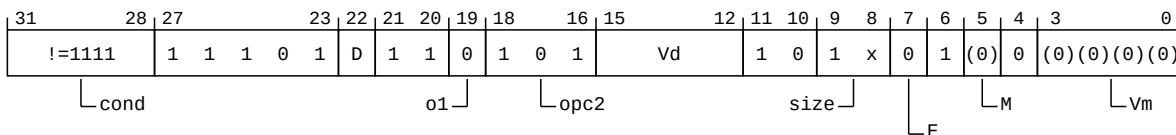
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as **NOP**. This means it behaves as if it fails the Condition code check.

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, #0.0

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, #0.0

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;

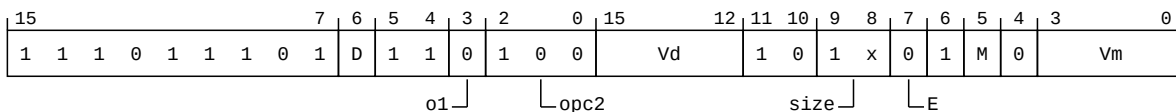
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

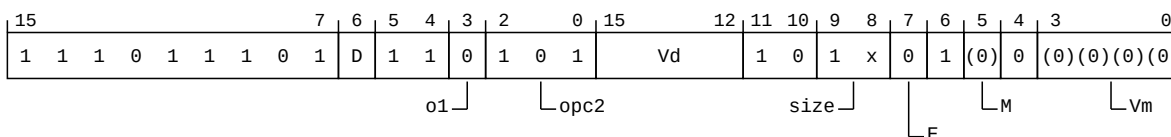
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMP{<c>} {<q>}.F32 <Sd>, #0.0

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, #0.0

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    var nzcvc : bits(4);
    case esize of
        when 32 =>
            let op32 : bits(32) = if with_zero then FPZero{32}('0') else S(m);
            nzcvc = FPCompare{32}(S(d), op32, quiet_nan_exc, fpcr);
        when 64 =>
            let op64 : bits(64) = if with_zero then FPZero{64}('0') else D(m);
            nzcvc = FPCompare{64}(D(d), op64, quiet_nan_exc, fpcr);
    end;

    FPSCR()[31:28] = nzcvc; // FPSCR.[N,Z,C,V] set to nzcvc
end;

```

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, >, or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2), and (Operand1 > Operand2) are false. An unordered comparison sets the [FPSCR](#) condition flags to N=0, Z=0, C=1, and V=1.

C6.1.52 VCMPE

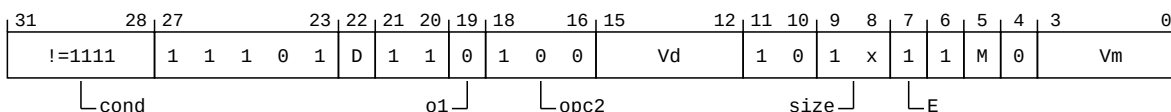
Vector Compare, raising Invalid Operation on NaN compares two floating-point registers, or one floating-point register and zero. It writes the result to the [FPSCR](#) flags. These are normally transferred to the [PSTATE](#).{N, Z, C, V} Condition flags by a subsequent [VMRS](#) instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

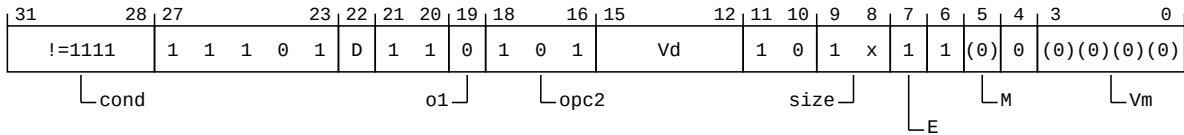
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, #0.0

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, #0.0

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer() = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;

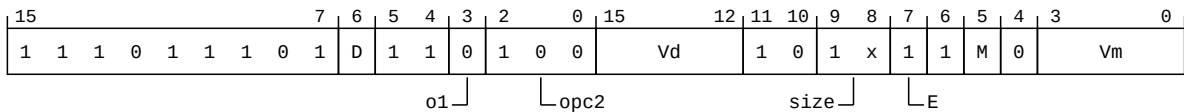
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

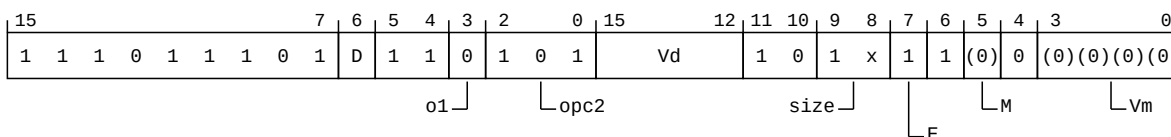
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let quiet_nan_exc : boolean = (E == '1');
let with_zero : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = ARBITRARY : integer;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as **NOP**. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    var nzcvc : bits(4);
    case esize of
        when 32 =>
            let op32 : bits(32) = if with_zero then FPZero{32}('0') else S(m);
            nzcvc = FPCompare{32}(S(d), op32, quiet_nan_exc, fpcr);
        when 64 =>
            let op64 : bits(64) = if with_zero then FPZero{64}('0') else D(m);
            nzcvc = FPCompare{64}(D(d), op64, quiet_nan_exc, fpcr);
    end;

    FPCR() [31:28] = nzcvc; // FPCR.[N,Z,C,V] set to nzcvc
end;

```

Operational Information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, >, or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2), and (Operand1 > Operand2) are false. An unordered comparison sets the [FPSCR](#) condition flags to N=0, Z=0, C=1, and V=1.

C6.1.53 VCNT

Vector Count Set Bits counts the number of bits that are one in each element in a vector, and places the results in a second vector.

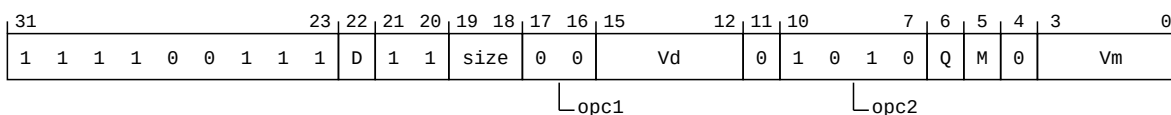
The operand vector elements must be 8-bit fields.

The result vector elements are 8-bit integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VCNT{<c>}{<q>}.8 <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VCNT{<c>}{<q>}.8 <Qd>, <Qm>

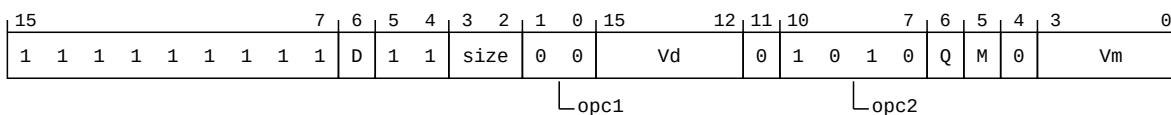
Decode for all variants of this encoding

```

if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8;
let elements : integer = 8;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if O == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VCNT{<c>}{<q>}.8 <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCNT{<c>}{<q>}.8 <Qd>, <Qm>

Decode for all variants of this encoding

```
if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8;
let elements : integer = 8;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = BitCount(D(m+r)[e*:esize])[esize-1:0];
        end;
    end;
end;
```

C6.1.54 VCVT (between double-precision and single-precision)

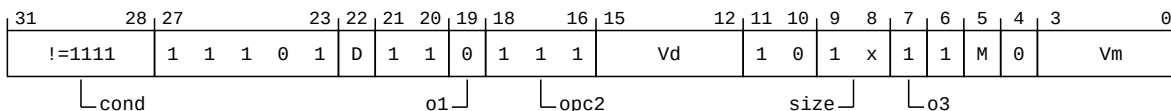
Convert between double-precision and single-precision does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision to double-precision variant

Applies when (size == 10)

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Encoding for the Double-precision to single-precision variant

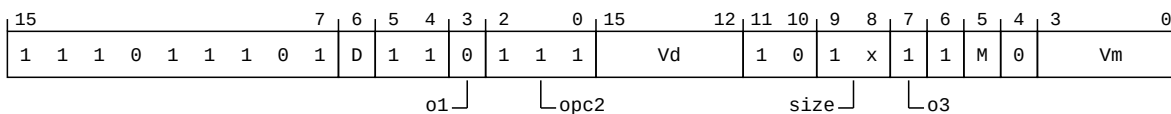
Applies when (size == 11)

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
let double_to_single : boolean = (size == '11');
let d : integer = if double_to_single then UInt(Vd::D) else UInt(D::Vd);
let m : integer = if double_to_single then UInt(M::Vm) else UInt(Vm::M);
```

T1



Encoding for the Single-precision to double-precision variant

Applies when (size == 10)

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Encoding for the Double-precision to single-precision variant

Applies when (size == 11)

VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
let double_to_single : boolean = (size == '11');
let d : integer = if double_to_single then UInt(Vd::D) else UInt(D::Vd);
let m : integer = if double_to_single then UInt(M::Vm) else UInt(Vm::M);
```

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    if double_to_single then
        S(d) = FPConvert{32, 64}(D(m), EffectiveFPCR());
    else
        D(d) = FPConvert{64, 32}(S(m), EffectiveFPCR());
    end;
end;
```

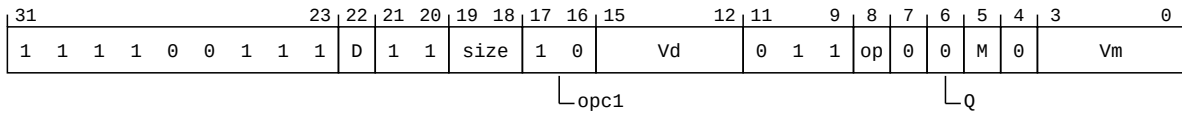
C6.1.55 VCVT (between half-precision and single-precision, Advanced SIMD)

Vector Convert between half-precision and single-precision converts each element in a vector from single-precision to half-precision floating-point, or from half-precision to single-precision, and places the results in a second vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Half-precision to single-precision variant

Applies when (op == 1)

VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 0)

VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm>

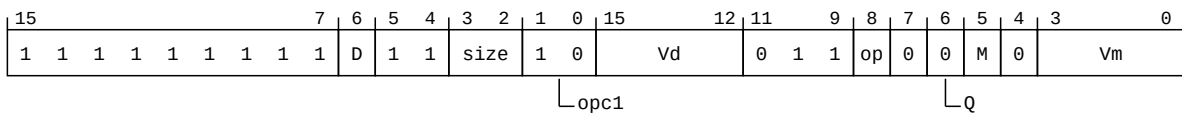
Decode for all variants of this encoding

```

if size != '01' then Undefined(); end;
if op == '1' && Vd[0] == '1' then Undefined(); end;
if op == '0' && Vm[0] == '1' then Undefined(); end;
let half_to_single : boolean = (op == '1');
let esize : integer{} = 16;
let elements : integer = 4;
let m : integer = UInt(M::Vm);
let d : integer = UInt(D::Vd);

```

T1



Encoding for the Half-precision to single-precision variant

Applies when (op == 1)

VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 0)

VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm>

Decode for all variants of this encoding

```
if size != '01' then Undefined(); end;
if op == '1' && Vd[0] == '1' then Undefined(); end;
if op == '0' && Vm[0] == '1' then Undefined(); end;
let half_to_single : boolean = (op == '1');
let esize : integer = 16;
let elements : integer = 4;
let m : integer = UInt(M::Vm);
let d : integer = UInt(D::Vd);
```

Assembler Symbols

<c>

For the “A1 Half-precision to single-precision” and “A1 Single-precision to half-precision” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Half-precision to single-precision” and “T1 Single-precision to half-precision” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for e = 0 to elements-1 do
        if half_to_single then
            Q(d>>1)[e*:32] = FPConvert{32, 16}(Din(m)[e*:16], fpcr);
        else
            D(d)[e*:16] = FPConvert{16, 32}(Qin(m>>1)[e*:32], fpcr);
        end;
    end;
end;
```

C6.1.56 VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	23	22	21	20	19	18	17	16	15	12	11	9	8	7	6	5	4	3	0				
1	1	1	1	0	0	1	1	1	D	1	1	size	1	1	Vd	0	1	1	op	Q	M	0	Vm

opc1

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let to_integer : boolean = (op[1] == '1');
let unsigned : boolean = (op[0] == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

1576543210									15121198765430																				
111111111									D	11		size		11		Vd			011			op		Q	M	0	Vm		
											└opc1																		

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let to_integer : boolean = (op[1] == '1');
let unsigned : boolean = (op[0] == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt1>

Is the data type for the elements of the destination vector, encoded in “size:op”:

size	op	<dt1>
01	0x	F16
01	10	S16
01	11	U16
10	0x	F32
10	10	S32
10	11	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size:op”:

size	op	<dt2>
01	00	S16
01	01	U16
01	1x	F16
10	00	S32
10	01	U32
10	1x	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    var result : bits(esize);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(m+r)[e*:esize];
            if to_integer then
                result = FPToFixed{esize, esize}(operand1, 0, unsigned, fpcr, FPRounding_ZERO);
            else
                result = FixedToFP{esize, esize}(operand1, 0, unsigned, fpcr, FPRounding_TIEEVEN);
            end;
            D(d+r)[e*:esize] = result;
        end;
    end;
end;
```


C6.1.57 VCVT (floating-point to integer, floating-point)

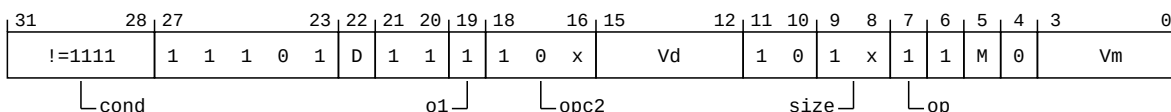
Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in a second register.

[VCVT \(between floating-point and fixed-point, floating-point\)](#) describes conversions between floating-point and 16-bit integers.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (opc2 == 100 && size == 10)

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (opc2 == 101 && size == 10)

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (opc2 == 100 && size == 11)

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Encoding for the Double-precision scalar variant

Applies when (opc2 == 101 && size == 11)

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

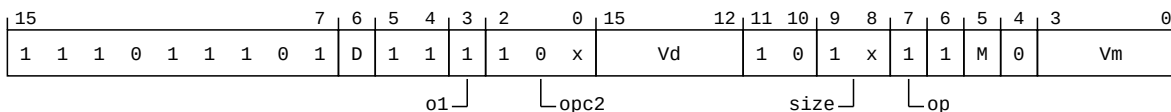
```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size != '1x' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when `(opc2 == 100 && size == 10)`

```
VCVT{<c>}{<q>}.U32.F32  <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 101 && size == 10)`

```
VCVT{<c>}{<q>}.S32.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 100 && size == 11)`

```
VCVT{<c>}{<q>}.U32.F64  <Sd>, <Dm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 101 && size == 11)`

```
VCVT{<c>}{<q>}.S32.F64  <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size != '1x' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Floating-point data-processing](#) for the T32 instruction set, or [Floating-point data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
end;

```

C6.1.58 VCVT (integer to floating-point, floating-point)

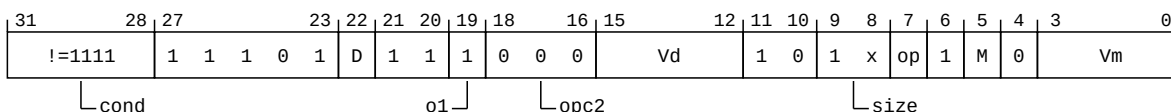
Convert integer to floating-point converts a 32-bit integer to floating-point using the rounding mode specified by the [FPSCR](#), and places the result in a second register.

[VCVT \(between floating-point and fixed-point, floating-point\)](#) describes conversions between floating-point and 16-bit integers.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

Decode for all variants of this encoding

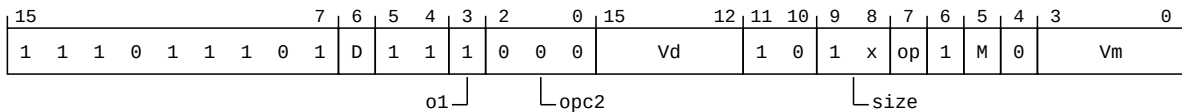
```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size != '1x' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

Decode for all variants of this encoding

```

if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size != '1x' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Floating-point data-processing](#) for the T32 instruction set, or [Floating-point data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the operand, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
```

C6.1.59 VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

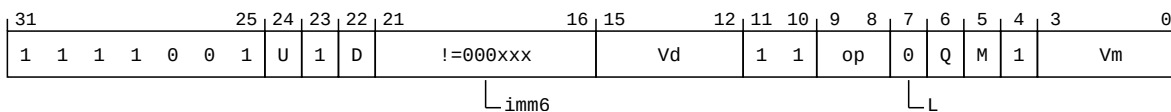
The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>

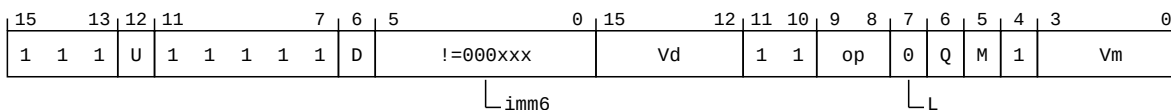
Decode for all variants of this encoding

```

if imm6 == '000xxx' then See("Related encodings"); end;
if op[1] == '0' then Undefined(); end;
if imm6 == '0xxxxx' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let to_fixed : boolean = (op[0] == '1');
let frac_bits : integer = 64 - UInt(imm6);
let unsigned : boolean = (U == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if op[1] == '0' then Undefined(); end;
if imm6 == '0xxxxx' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let to_fixed : boolean = (op[0] == '1');
let frac_bits : integer = 64 - UInt(imm6);
let unsigned : boolean = (U == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt1>

Is the data type for the elements of the destination vector, encoded in “op:U”:

op	U	<dt1>
00	x	F16
01	0	S16
01	1	U16
10	x	F32
11	0	S32
11	1	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “op:U”:

op	U	<dt2>
00	0	S16
00	1	U16
01	x	F16
10	0	S32
10	1	U32
11	x	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<fbits>

The number of fraction bits in the fixed point number, in the range 1 to 32 for 32-bit elements, or in the range 1 to 16 for 16-bit elements:

- (64 - <fbits>) is encoded in imm6.

An assembler can permit an <fbits> value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see [VCVT \(between floating-point and integer, Advanced SIMD\)](#).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : bits(esize);
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(m+r)[e*:esize];
            if to_fixed then
                result = FPToFixed{esize, esize}(operand1, frac_bits, unsigned, fpcr,
                                                    FPRounding_ZERO);
            else
                result = FixedToFP{esize, esize}(operand1, frac_bits, unsigned, fpcr,
                                                    FPRounding_TIEEVEN);
            end;
            D(d+r)[e*:esize] = result;
        end;
    end;
end;

```

C6.1.60 VCVT (between floating-point and fixed-point, floating-point)

Convert between floating-point and fixed-point converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point. Software can specify the fixed-point value as either signed or unsigned.

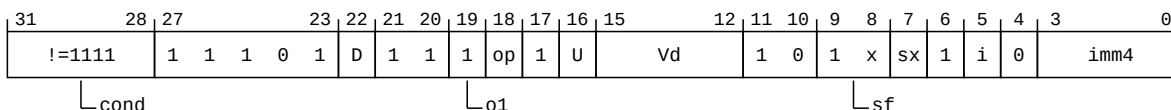
The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Single-precision scalar variant

Applies when (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when (op == 0 && sf == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when (op == 1 && sf == 11)

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

Decode for all variants of this encoding

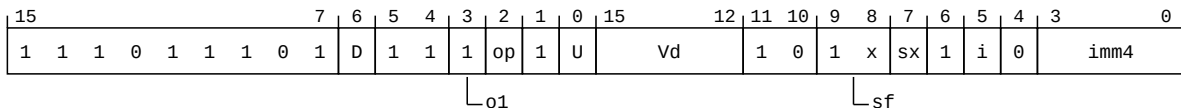
```
if sf != '1x' then Undefined(); end;
let to_fixed : boolean = (op == '1');
let unsigned : boolean = (U == '1');
let size : integer{} = if sx == '0' then 16 else 32;
let frac_bits : integer = size - UInt(imm4::i);
let fp_size : integer = 8 << UInt(sf);
let d : integer = if sf == '11' then UInt(D::Vd) else UInt(Vd::D);
if frac_bits < 0 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

T1



Encoding for the Single-precision scalar variant

Applies when (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Single-precision scalar variant

Applies when (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32    <Sdm>, <Sdm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when (op == 0 && sf == 11)

```
VCVT{<c>}{<g>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Encoding for the Double-precision scalar variant

Applies when (op == 1 && sf == 11)

```
VCVT{<c>}{<q>}.<dt>.F64    <Ddm>, <Ddm>, #<fbits>
```

Decode for all variants of this encoding

```

if sf != 'lx' then Undefined(); end;
let to_fixed : boolean = (op == '1');
let unsigned : boolean = (U == '1');
let size : integer{} = if sx == '0' then 16 else 32;
let frac_bits : integer = size - UInt(imm4::i);
let fp_size : integer = 8 << UInt(sf);
let d : integer = if sf == '11' then UInt(D::Vd) else UInt(Vd::D);
if frac_bits < 0 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `frac bits < 0`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VCVT \(between floating-point and fixed-point\)](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the fixed-point number, encoded in “U:sx”:

U	sx	<dt>
0	0	S16
0	1	S32
1	0	U16
1	1	U32

<Sdm>

Is the 32-bit name of the SIMD&FP destination and source register, encoded in the “Vd:D” field.

<fbits>

The number of fraction bits in the fixed-point number:

- If <dt> is S16 or U16, <fbits> must be in the range 0-16. (16 - <fbits>) is encoded in [imm4, i]
- If <dt> is S32 or U32, <fbits> must be in the range 1-32. (32 - <fbits>) is encoded in [imm4, i].

<Ddm>

Is the 64-bit name of the SIMD&FP destination and source register, encoded in the “D:Vd” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if to_fixed then
        var result : bits(size);
        case fp_size of
            when 32 =>
                result = FPToFixed{size, 32}(S(d), frac_bits, unsigned, fpcr, FPRounding_ZERO);
                S(d) = Extend{32}(result, unsigned);
            when 64 =>
                result = FPToFixed{size, 64}(D(d), frac_bits, unsigned, fpcr, FPRounding_ZERO);
                D(d) = Extend{64}(result, unsigned);
        end;
    else
        case fp_size of
            when 32 =>
                S(d) = FixedToFP{32, size}(S(d)[size-1:0], frac_bits, unsigned, fpcr,
                    FPRounding_TIEEVEN);
            when 64 =>
                D(d) = FixedToFP{64, size}(D(d)[size-1:0], frac_bits, unsigned, fpcr,
                    FPRounding_TIEEVEN);
        end;
    end;
end;
```

C6.1.61 VCVTA (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest with Ties to Away

Vector Convert floating-point to integer with Round to Nearest with Ties to Away converts each element in a vector from floating-point to integer using the Round to Nearest with Ties to Away rounding mode, and places the results in a second vector.

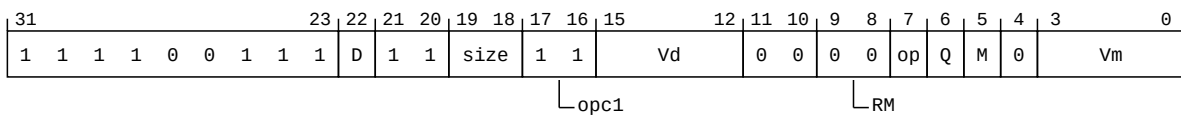
The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VCVTA{<α>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when $(Q == 1)$

VCVTA{<α>}.<dt>.<dt2> <Qd>, <Qm>

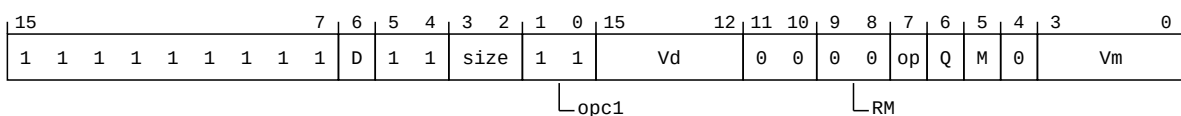
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPDecoderRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
  for e = 0 to elements-1 do
    D(d+r)[e*:esize] = FPToFixed{esize, esize}(D(m+r)[e*:esize],
                                                    0, unsigned, fpcr, rounding);
  end;
end;
```

C6.1.62 VCVTA (floating-point)

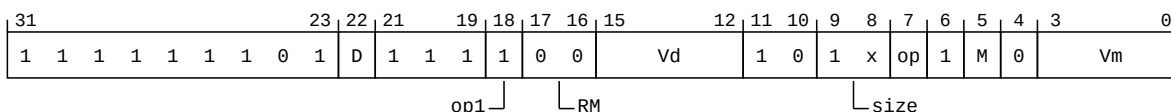
Convert floating-point to integer with Round to Nearest with Ties to Away

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

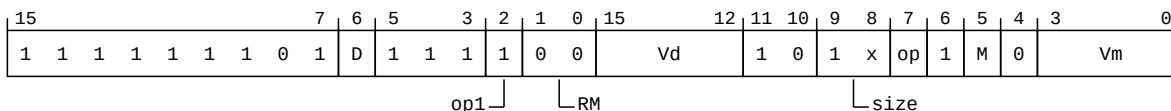
Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer() = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 32 =>
    S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
  when 64 =>
    S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```

C6.1.63 VCVTB

Convert to or from a half-precision value in the bottom half of a single-precision register

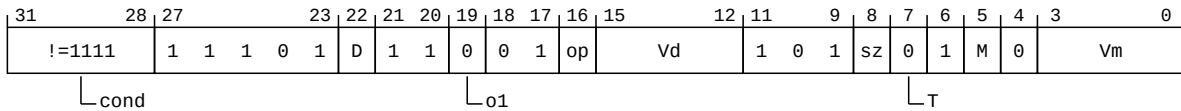
Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Half-precision to single-precision variant

Applies when (op == 0 && sz == 0)

VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Encoding for the Half-precision to double-precision variant

Applies when (op == 0 && sz == 1)

VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 1 && sz == 0)

VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Encoding for the Double-precision to half-precision variant

Applies when (op == 1 && sz == 1)

VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>

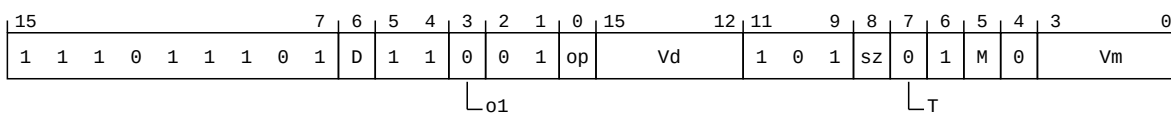
Decode for all variants of this encoding

```

let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);

```

T1



Encoding for the Half-precision to single-precision variant

Applies when (op == 0 && sz == 0)

VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Encoding for the Half-precision to double-precision variant

Applies when (op == 0 && sz == 1)

VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 1 && sz == 0)

VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Encoding for the Double-precision to half-precision variant

Applies when (op == 1 && sz == 1)

VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```

let d : integer = if sz == 'l' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == 'l' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == 'l');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == 'l' then 16 else 0);

```

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var hp : bits(16);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if convert_from_half then
        hp = S(m)[lowbit+15:lowbit];
        if uses_double then
            D(d) = FPConvert{64, 16}(hp, fpcr);
        else
            S(d) = FPConvert{32, 16}(hp, fpcr);
        end;
    else
        if uses_double then
            hp = FPConvert{16, 64}(D(m), fpcr);
        else
            hp = FPConvert{16, 32}(S(m), fpcr);
        end;
        S(d)[lowbit+15:lowbit] = hp;
    end;
end;
```

C6.1.64 VCVTM (Advanced SIMD)

Vector Convert floating-point to integer with Round towards -Infinity converts each element in a vector from floating-point to integer using the Round towards -Infinity rounding mode, and places the results in a second vector.

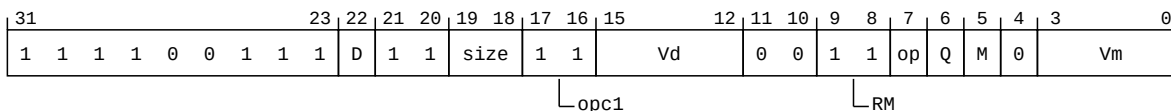
The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

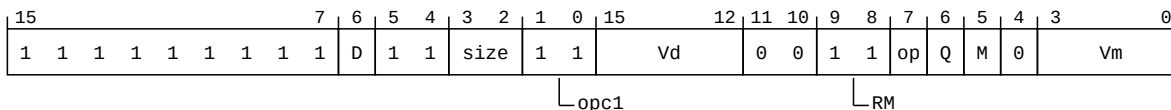
Applies when (Q == 1)

```
VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*:esize] = FPToFixed{esize, esize}(D(m+r)[e*:esize],
                                                    0, unsigned, fpcr, rounding);
    end;
end;
```

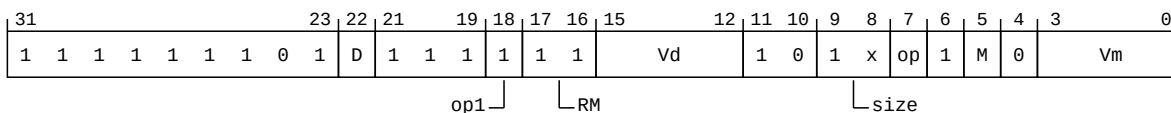
C6.1.65 VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>

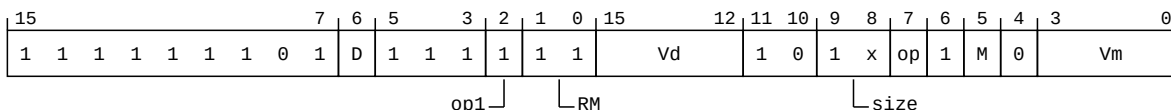
Decode for all variants of this encoding

```

if size != 'lx' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 32 =>
    S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
  when 64 =>
    S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```

C6.1.66 VCVTN (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector.

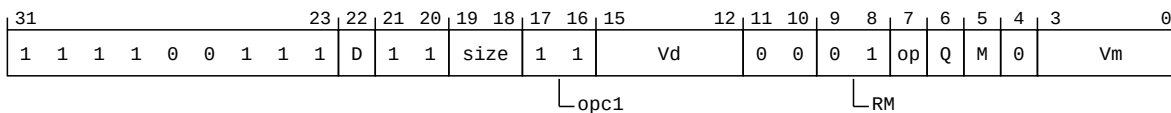
The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>

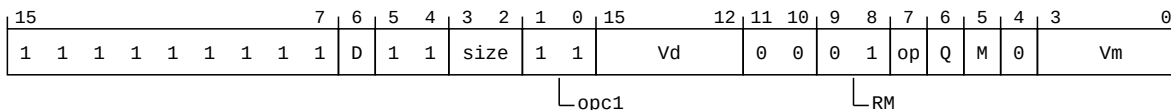
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer() = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();  
CheckAdvSIMDEnabled();  
let fpcr : FPCR_Type = StandardFPCR();  
for r = 0 to regs-1 do  
    for e = 0 to elements-1 do  
        D(d+r)[e*:esize] = FPToFixed{esize, esize}(D(m+r)[e*:esize],  
                                                    0, unsigned, fpcr, rounding);  
    end;  
end;
```

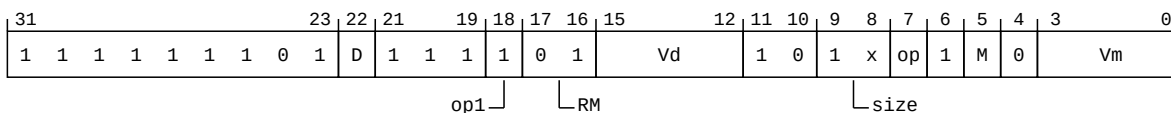
C6.1.67 VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register.

Depending on settings in the **CPACR**, **NSACR**, **HCPTR**, and **FPEXC** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>

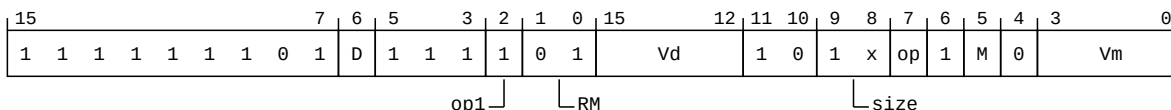
Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '1l' then UInt(M::Vm) else UInt(Vm::M);

```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTN{<α>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 32 =>
    S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
  when 64 =>
    S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```

C6.1.68 VCVTP (Advanced SIMD)

Vector Convert floating-point to integer with Round towards +Infinity converts each element in a vector from floating-point to integer using the Round towards +Infinity rounding mode, and places the results in a second vector.

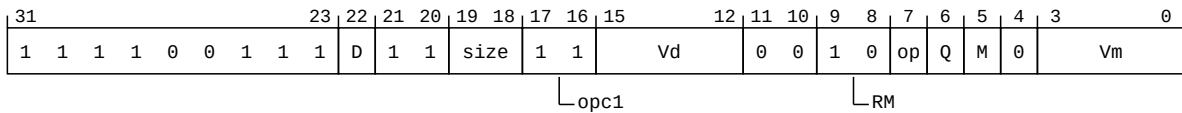
The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>

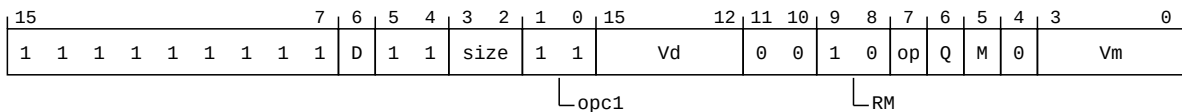
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2>

Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
let fpcr : FPCR_Type = StandardFPCR();
for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        D(d+r)[e*:esize] = FPToFixed{esize, esize}(D(m+r)[e*:esize],
                                                    0, unsigned, fpcr, rounding);
    end;
end;
```

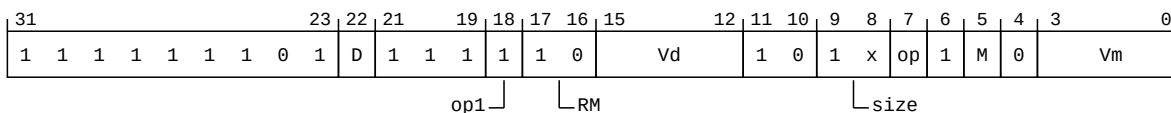
C6.1.69 VCVTP (floating-point)

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>

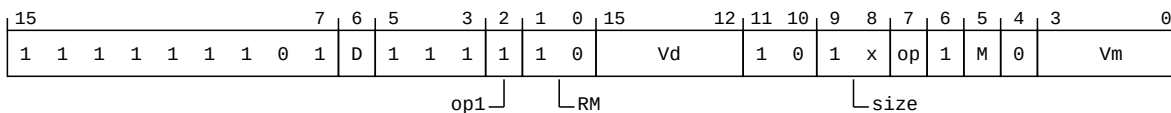
Decode for all variants of this encoding

```

if size != 'lx' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let unsigned : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 32 =>
    S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
  when 64 =>
    S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
end;
```

C6.1.70 VCVTR

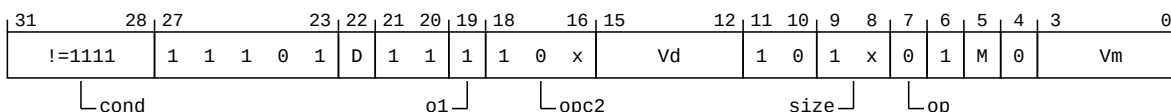
Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the [FPSCR](#) and places the result in a second register.

[VCVT \(between floating-point and fixed-point, floating-point\)](#) describes conversions between floating-point and 16-bit integers.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (opc2 == 100 && size == 10)

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (opc2 == 101 && size == 10)

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (opc2 == 100 && size == 11)

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Encoding for the Double-precision scalar variant

Applies when (opc2 == 101 && size == 11)

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

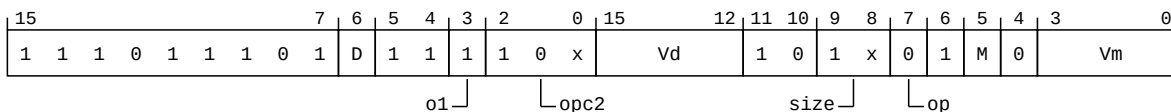
```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size != '1x' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when `(opc2 == 100 && size == 10)`

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when `(opc2 == 101 && size == 10)`

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 100 && size == 11)`

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Encoding for the Double-precision scalar variant

Applies when `(opc2 == 101 && size == 11)`

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && opc2 != '10x' then See("Related encodings"); end;
if size != '1x' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let m : integer = if size == '11' && opc2[2] == '1' then UInt(M::Vm) else UInt(Vm::M);
let d : integer = if size == '11' && opc2[2] == '0' then UInt(D::Vd) else UInt(Vd::D);
let to_integer : boolean = (opc2[2] == '1');
let unsigned : boolean = (if to_integer then opc2[0] else op) == '0';
let zero_rounding : boolean = to_integer && op == '1';
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Floating-point data-processing](#) for the T32 instruction set, or [Floating-point data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 32 =>
                S(d) = FPToFixed{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                S(d) = FPToFixed{32, 64}(D(m), 0, unsigned, fpcr, rounding);
        end;
    else
        case esize of
            when 32 =>
                S(d) = FixedToFP{32, 32}(S(m), 0, unsigned, fpcr, rounding);
            when 64 =>
                D(d) = FixedToFP{64, 32}(S(m), 0, unsigned, fpcr, rounding);
        end;
    end;
end;
end;

```

C6.1.71 VCVTT

Convert to or from a half-precision value in the top half of a single-precision register

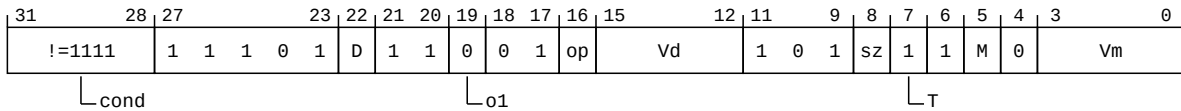
Convert to or from a half-precision value in the top half of a single-precision register does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Half-precision to single-precision variant

Applies when (op == 0 && sz == 0)

VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Encoding for the Half-precision to double-precision variant

Applies when (op == 0 && sz == 1)

VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 1 && sz == 0)

VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Encoding for the Double-precision to half-precision variant

Applies when (op == 1 && sz == 1)

VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>

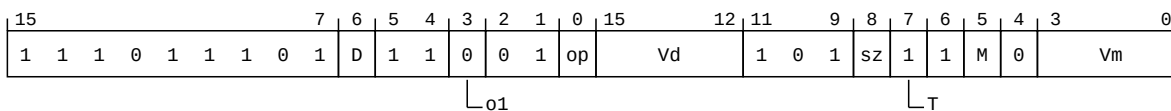
Decode for all variants of this encoding

```

let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);

```

T1



Encoding for the Half-precision to single-precision variant

Applies when (op == 0 && sz == 0)

VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Encoding for the Half-precision to double-precision variant

Applies when (op == 0 && sz == 1)

VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Encoding for the Single-precision to half-precision variant

Applies when (op == 1 && sz == 0)

VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Encoding for the Double-precision to half-precision variant

Applies when (op == 1 && sz == 1)

VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>

Decode for all variants of this encoding

```
let d : integer = if sz == '1' && op == '0' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if sz == '1' && op == '1' then UInt(M::Vm) else UInt(Vm::M);
let uses_double : boolean = (sz == '1');
let convert_from_half : boolean = (op == '0');
let lowbit : integer{} = (if T == '1' then 16 else 0);
```

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var hp : bits(16);
    let fpcr : FPCR_Type = EffectiveFPCR();
    if convert_from_half then
        hp = S(m)[lowbit+15:lowbit];
        if uses_double then
            D(d) = FPConvert{64, 16}(hp, fpcr);
        else
            S(d) = FPConvert{32, 16}(hp, fpcr);
        end;
    else
        if uses_double then
            hp = FPConvert{16, 64}(D(m), fpcr);
        else
            hp = FPConvert{16, 32}(S(m), fpcr);
        end;
        S(d)[lowbit+15:lowbit] = hp;
    end;
end;
```

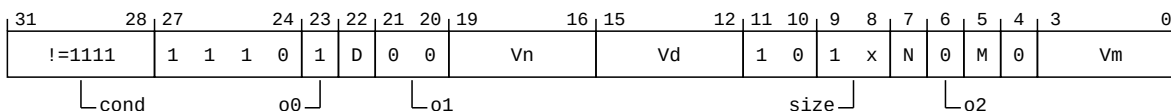
C6.1.72 VDIV

Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VDIV{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VDIV{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

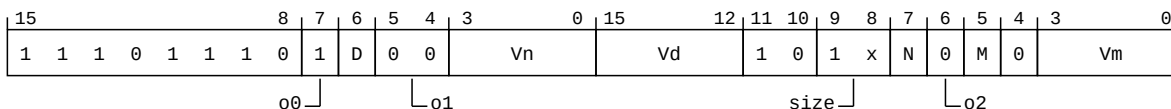
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VDIV{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VDIV{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 =>
            S(d) = FPDIV{32}(S(n), S(m), fpcr);
        when 64 =>
            D(d) = FPDIV{64}(D(n), D(m), fpcr);
    end;
end;
```

C6.1.73 VDUP (general-purpose register)

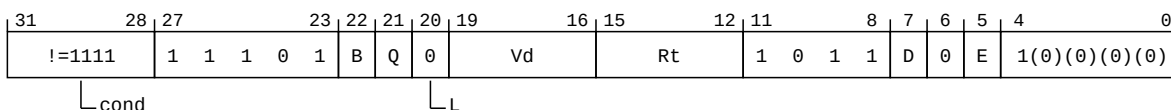
Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VDUP{<c>}{<q>}.<size> <Dd>, <Rt>
```

Encoding for the 128-bit SIMD vector variant

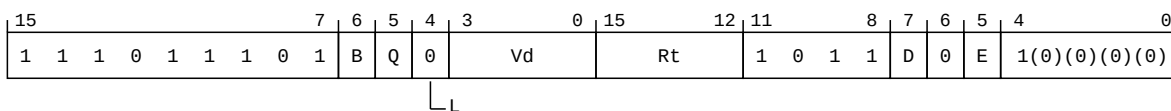
Applies when (Q == 1)

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt>
```

Decode for all variants of this encoding

```
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
let regs : integer = if Q == '0' then 1 else 2;
if B::E == '11' then Undefined(); end;
let esize : integer{} = 32 >> UInt(B::E);
let elements : integer = 64 DIV esize;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VDUP{<c>}{<q>}.<size> <Dd>, <Rt>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt>
```

Decode for all variants of this encoding

```
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
let regs : integer = if Q == '0' then 1 else 2;
if B::E == '11' then Undefined(); end;
let esize : integer{} = 32 >> UInt(B::E);
let elements : integer = 64 DIV esize;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#). Arm strongly recommends that any VDUP instruction is unconditional, see [Conditional execution](#).

<q>

See [Standard assembler syntax fields](#).

<size>

The data size for the elements of the destination vector. It must be one of:

- 8** Encoded as [b, e] = 0b10.
- 16** Encoded as [b, e] = 0b01.
- 32** Encoded as [b, e] = 0b00.

<Dd>

The destination vector for a doubleword operation.

<Rt>

The Arm source register.

<Qd>

The destination vector for a quadword operation.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let scalar : bits(esize) = R(t)[esize-1:0];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*esize] = scalar;
        end;
    end;
end;
```

C6.1.74 VDUP (scalar)

Duplicate vector element to vector duplicates a single element of a vector into every element of the destination vector.

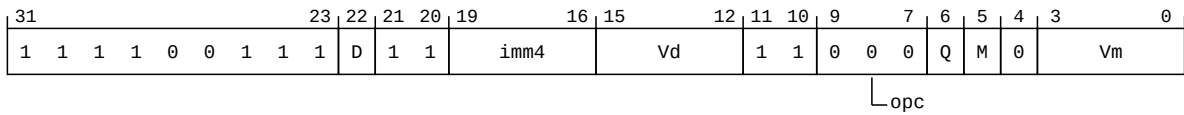
The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit fields. There is no distinction between data types.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

Applies when (Q == 0)

VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>

Encoding

Applies when (Q == 1)

VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>

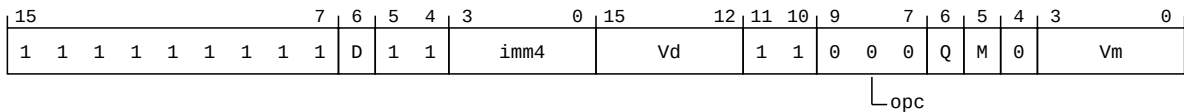
Decode for all variants of this encoding

```

if imm4 == 'x000' then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let lsb : integer{} = LowestSetBit(imm4[2:0]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(imm4[3:lsb+1]);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer{} = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding

Applies when (Q == 0)

VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>

Encoding

Applies when (Q == 1)

VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>

Decode for all variants of this encoding

```
if imm4 == 'x000' then Undefined(); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let lsb : integer{} = LowestSetBit(imm4[2:0]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(imm4[3:lsb+1]);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let elements : integer{} = 64 DIV esize;
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 Advanced SIMD duplicate (scalar)” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Advanced SIMD duplicate (scalar)” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

The data size. It must be one of:

- 8** Encoded as imm4<0> = ‘1’. imm4<3:1> encodes the index [x] of the scalar.
- 16** Encoded as imm4<1:0> = ‘10’. imm4<3:2> encodes the index [x] of the scalar.
- 32** Encoded as imm4<2:0> = ‘100’. imm4<3> encodes the index [x] of the scalar.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm[x]>

The scalar. For details of how [x] is encoded, see the description of <size>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let scalar : bits(esize) = D(m)[index*esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*esize] = scalar;
        end;
    end;
end;
```

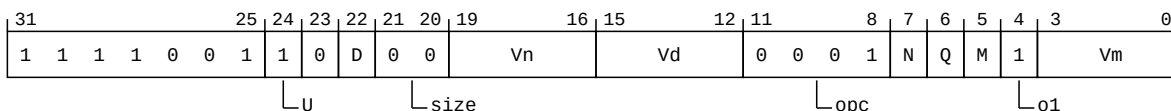
C6.1.75 VEOR

Vector Bitwise Exclusive-OR performs a bitwise exclusive-OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VEOR{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VEOR{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

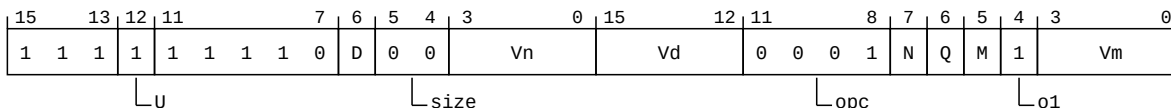
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VEOR{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VEOR{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) XOR D(m+r);
    end;
end;

```

C6.1.76 VEXT (byte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

The elements of the vectors are treated as being 8-bit fields. There is no distinction between data types.

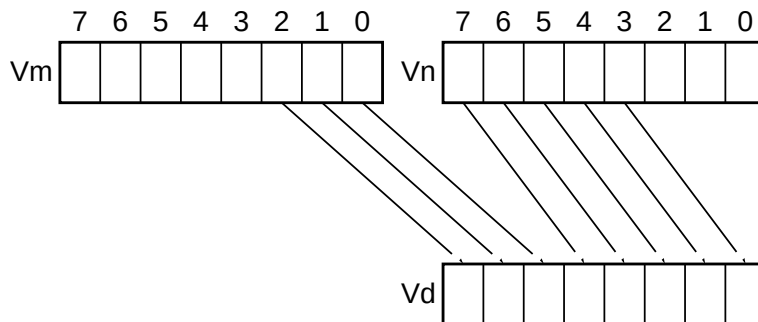


Figure C6-1 VEXT doubleword operation for imm = 3

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VEXT \(multibyte elements\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31									23	22	21	20	19		16	15			12	11		8	7	6	5	4	3	0
1	1	1	1	0	0	1	0	1	D	1	1		Vn			Vd			imm4			N	Q	M	0		Vm	

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VEXT{<c>}{<q>}.8 {<Dd>, }<Dn>, <Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

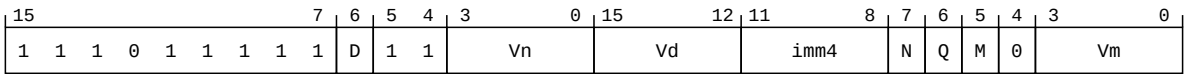
Applies when (Q == 1)

VEXT{<c>}{<q>}.8 {<Qd>, }<Qn>, <Qm>, #<imm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if Q == '0' && imm4[3] == '1' then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let position : integer{} = 8 * UInt(imm4);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VEXT{<c>}{<q>}.8 {<Dd>, }<Dn>, <Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VEXT{<c>}{<q>}.8 {<Qd>, }<Qn>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if Q == '0' && imm4[3] == '1' then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let position : integer{} = 8 * UInt(imm4);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 7, encoded in the “imm4” field.

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 15, encoded in the “imm4” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        Q(d>>1) = (Q(m>>1)::Q(n>>1))[position+127:position];
    else
        D(d) = (D(m)::D(n))[position+63:position];
    end;
end;
```

C6.1.77 VEXT (multibyte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

This is a pseudo-instruction of [VEXT \(byte elements\)](#). This means:

- The encodings in this description are named to match the encodings of [VEXT \(byte elements\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VEXT \(byte elements\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31								23	22	21	20	19	16		15	12		11	8		7	6	5	4	3	0
1	1	1	1	0	0	1	0	1	D	1	1	Vn		Vd		imm4			N	Q	M	0	Vm			

Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VEXT{<c>}{<q>}.<size> {<Dd>, }<Dn>, <Dm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Dd>, }<Dn>, <Dm>, #<imm*(size/8)>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VEXT{<c>}{<q>}.<size> {<Qd>, }<Qn>, <Qm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Qd>, }<Qn>, <Qm>, #<imm*(size/8)>

T1

15									7		6	5	4	3	0		15			12		11	8		7	6	5	4	3	0	
1	1	1	0	1	1	1	1	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VEXT{<c>}{<q>}.<size> {<Dd>, }<Dn>, <Dm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Dd>, }<Dn>, <Dm>, #<imm*(size/8)>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$VEXT\{<c>\}\{<q>\}.<size>\{<Qd>, \}<Qn>, <Qm>, \#<imm>$

is equivalent to

$VEXT\{<c>\}\{<q>\}.8\{<Qd>, \}<Qn>, <Qm>, \#<imm>(size/8)>$

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the size of the operation, and can be one of 16 or 32.

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the size of the operation, and can be one of 16, 32, or 64.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to $(128/<size>)-1$.

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to $(64/<size>)-1$.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as $<Qd>*2$.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as $<Qn>*2$.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as $<Qm>*2$.

Operation

The description of [VEXT \(byte elements\)](#) gives the operational pseudocode for this instruction.

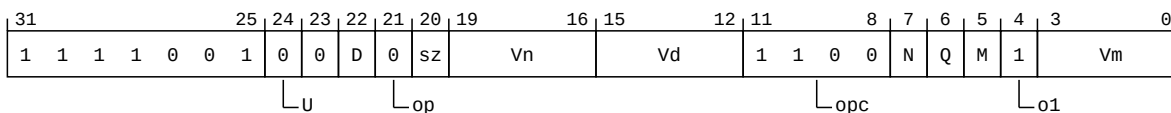
C6.1.78 VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

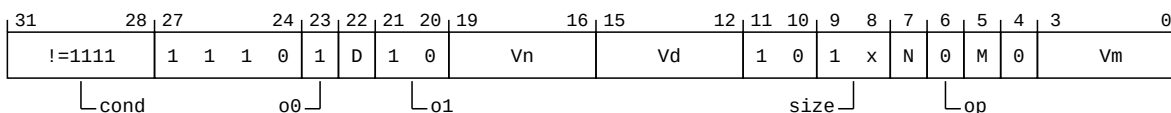
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

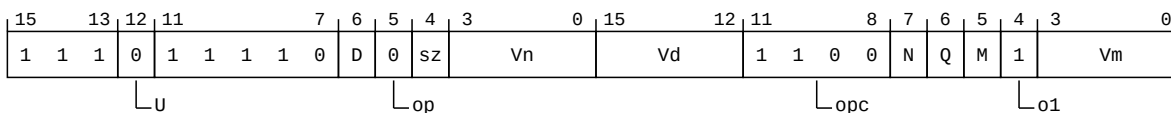
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

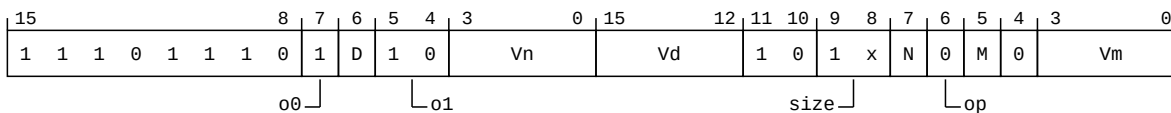
```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

`VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

`VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>`

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var oplelt : bits(esize) = D(n+r)[e*:esize];
                if opl_neg then oplelt = FPNeg{esize}(oplelt, fpcr); end;
                D(d+r)[e*:esize] = FPMulAdd{esize}(D(d+r)[e*:esize],
                                                    oplelt, D(m+r)[e*:esize], fpcr);
            end;
        end;

    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 32 =>
                let op32 : bits(32) = if opl_neg then FPNeg{32}(S(n), fpcr) else S(n);
                S(d) = FPMulAdd{32}(S(d), op32, S(m), fpcr);
            when 64 =>
                let op64 : bits(64) = if opl_neg then FPNeg{64}(D(n), fpcr) else D(n);
                D(d) = FPMulAdd{64}(D(d), op64, D(m), fpcr);
        end;
    end;
end;

```

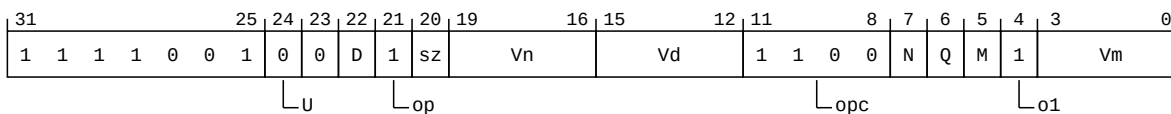
C6.1.79 VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

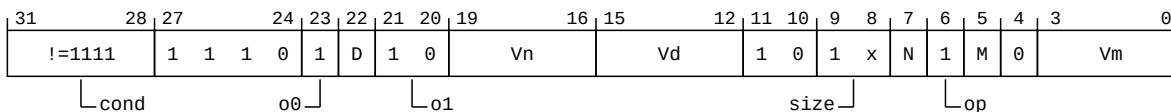
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

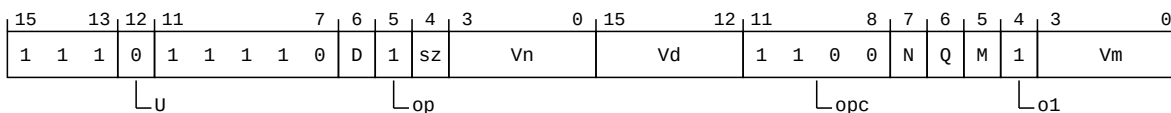
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

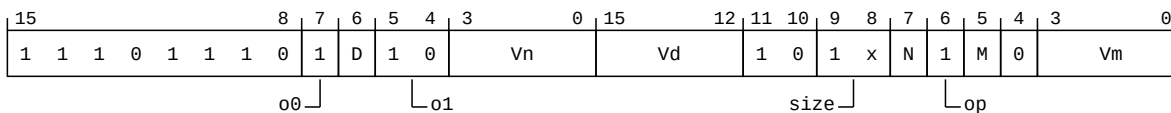
```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var oplelt : bits(esize) = D(n+r)[e*:esize];
                if opl_neg then oplelt = FPNeg{esize}(oplelt, fpcr); end;
                D(d+r)[e*:esize] = FPMulAdd{esize}(D(d+r)[e*:esize],
                                                    oplelt, D(m+r)[e*:esize], fpcr);
            end;
        end;

    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 32 =>
                let op32 : bits(32) = if opl_neg then FPNeg{32}(S(n), fpcr) else S(n);
                S(d) = FPMulAdd{32}(S(d), op32, S(m), fpcr);
            when 64 =>
                let op64 : bits(64) = if opl_neg then FPNeg{64}(D(n), fpcr) else D(n);
                D(d) = FPMulAdd{64}(D(d), op64, D(m), fpcr);
        end;
    end;
end;

```

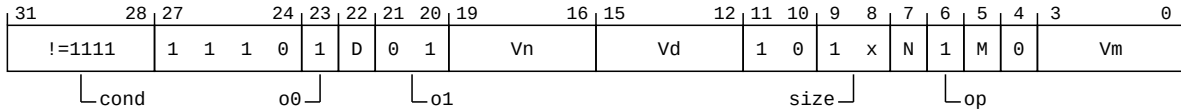
C6.1.80 VFNMA

Vector Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

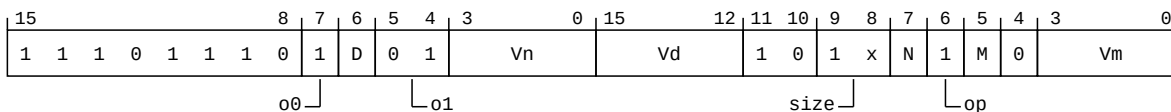
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 =>
            let op32 : bits(32) = if op1_neg then FPNeg{32}(S(n), fpcr) else S(n);
            S(d) = FPMulAdd{32}(FPNeg{32}(S(d), fpcr), op32, S(m), fpcr);
        when 64 =>
            let op64 : bits(64) = if op1_neg then FPNeg{64}(D(n), fpcr) else D(n);
            D(d) = FPMulAdd{64}(FPNeg{64}(D(d), fpcr), op64, D(m), fpcr);
    end;
end;
```

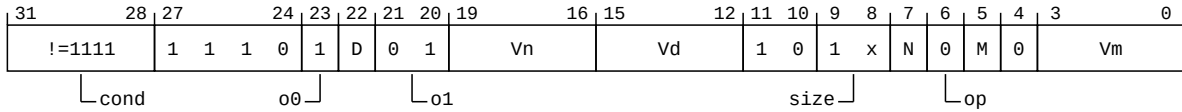
C6.1.81 VFNMS

Vector Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

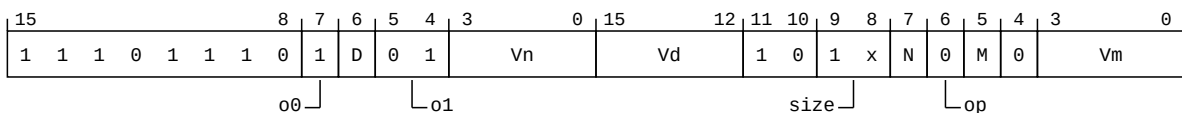
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let opl_neg : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let opl_neg : boolean = (op == '1');
let esize : integer() = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 =>
            let op32 : bits(32) = if opl_neg then FPNeg{32}(S(n), fpcr) else S(n);
            S(d) = FPMulAdd{32}(FPNeg{32}(S(d), fpcr), op32, S(m), fpcr);
        when 64 =>
            let op64 : bits(64) = if opl_neg then FPNeg{64}(D(n), fpcr) else D(n);
            D(d) = FPMulAdd{64}(FPNeg{64}(D(d), fpcr), op64, D(m), fpcr);
    end;
end;
```

C6.1.82 VHADD

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. For rounded results, see [VRHADD](#)).

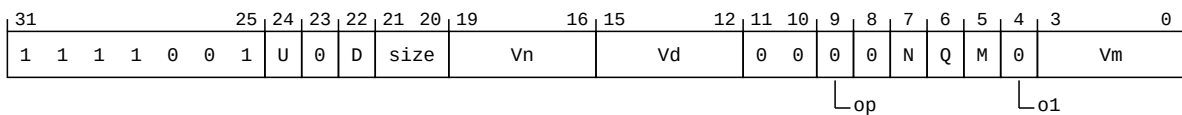
The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

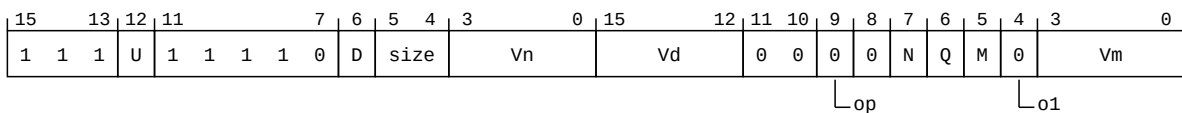
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if add then element1+element2 else element1-element2) >> 1;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;

```

C6.1.83 VHSUB

Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. There is no rounding version.

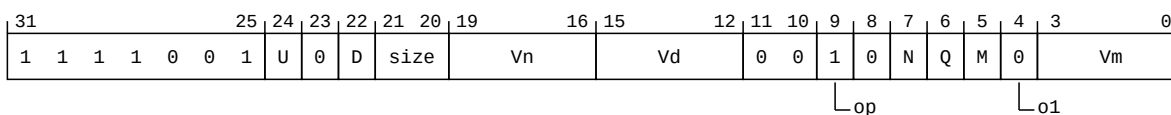
The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

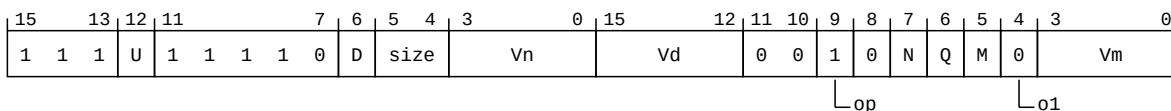
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let add : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer{} = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if add then element1+element2 else element1-element2) >> 1;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

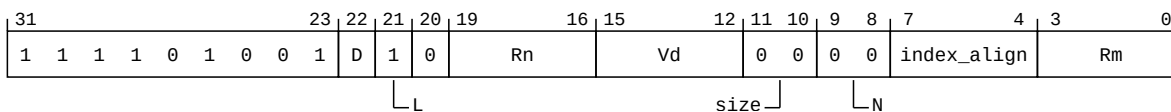
C6.1.84 VLD1 (single element to one lane)

Load single 1-element structure to one lane of one register loads one element from memory into one element of a register. Elements of the register that are not loaded are unchanged. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#), [A2](#), and [A3](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

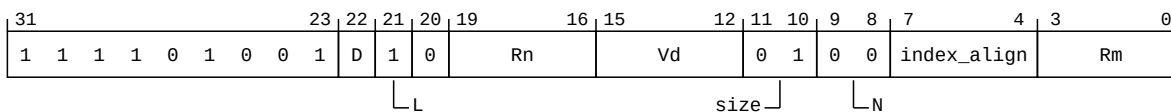
Decode for all variants of this encoding

```

if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

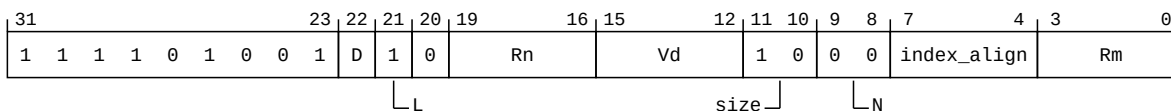
Decode for all variants of this encoding

```

if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VLD1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \langle \text{list} \rangle, [\langle \text{Rn} \rangle\{\langle \text{align} \rangle\}], \langle \text{Rm} \rangle$$

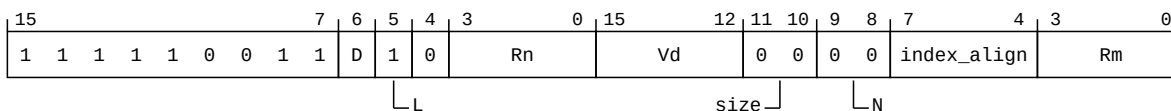
Decode for all variants of this encoding

```

if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D:Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

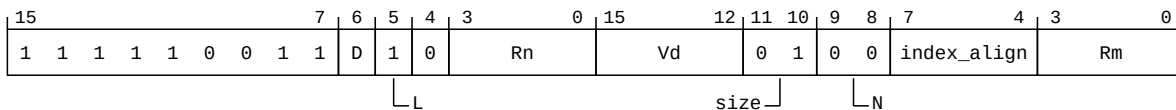
Decode for all variants of this encoding

```

if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```


T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

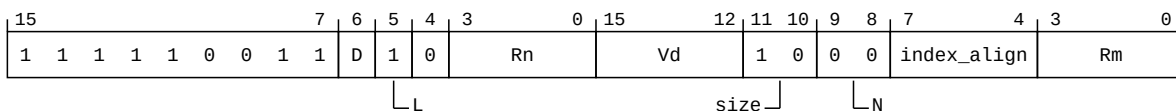
$$\text{VLD1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$
Decode for all variants of this encoding

```

if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D:Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then See("VLD1 (single element to all lanes)"); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the single 64-bit name of the SIMD&FP register holding the element.

The list must be { <Dd>[<index>] }.

The register <Dd> is encoded in the “D:Vd” field.

The permitted values and encoding of <index> depend on <size>:

<size> == **8** <index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.

<size> == **16** <index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.

<size> == **32** <index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

When <size> = 8, <align> must be omitted, otherwise it is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> = 8 Encoded in the “index_align<0>” field as 0.
<size> = 16 Encoded in the “index_align<1:0>” field as 0b00.
<size> = 32 Encoded in the “index_align<2:0>” field as 0b000.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> = 16 <align> is 16, meaning 16-bit alignment, encoded in the “index_align<1:0>” field as 0b01.
<size> = 32 <align> is 32, meaning 32-bit alignment, encoded in the “index_align<2:0>” field as 0b011.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    D(d)[index:(8*ebytes)] = MemU{8*ebytes}(address);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
end;

```

C6.1.85 VLD1 (single element to all lanes)

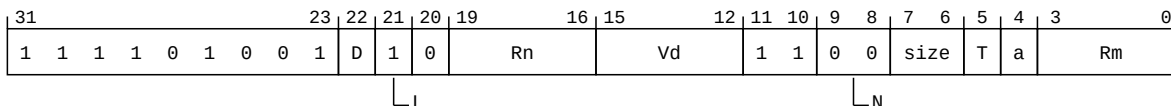
Load single 1-element structure and replicate to all lanes of one register

Load single 1-element structure and replicate to all lanes of one register loads one element from memory into every element of one or two vectors. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VLD1}\{\langle c \rangle\}\{\langle g \rangle\}.\langle \text{size} \rangle \langle \text{list} \rangle, [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if size == '11' || (size == '00' && a == '1') then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let regs : integer = if T == '0' then 1 else 2;
let alignment : integer{} = if a == '0' then 1 else ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

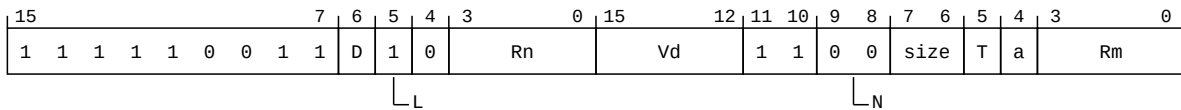
```

CONSTRAINED UNPREDICTABLE behavior

If $d_{\text{+regs}} > 32$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VLD1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if size == '11' || (size == '00' && a == '1') then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let regs : integer = if T == '0' then 1 else 2;
let alignment : integer{} = if a == '0' then 1 else ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $d_{\text{regs}} > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly **VLD1** (single element to all lanes).

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd>[] } Encoded in the “T” field as 0.

{ <Dd>[], <Dd+1>[] } Encoded in the “T” field as 1.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

When <size> == 8, <align> must be omitted, otherwise it is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “a” field as 0.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16 <align> is 16, meaning 16-bit alignment, encoded in the “a” field as 1.

<size> == 32 <align> is 32, meaning 32-bit alignment, encoded in the “a” field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != ELO;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    let esize : integer{} = 8 * ebytes;
    let element : bits(esize) = MemU{8*ebytes}(address);
    let replicated_element : bits(64) = Replicate{}(element);
    for r = 0 to regs-1 do
        D(d+r) = replicated_element;
    end;
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
```

C6.1.86 VLD1 (multiple single elements)

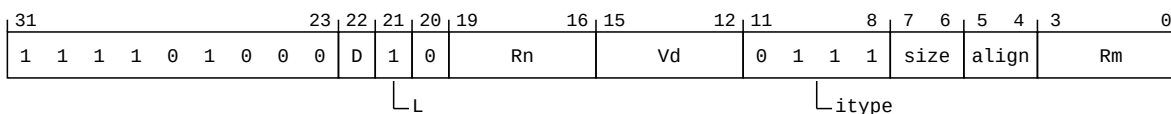
Load multiple single 1-element structures to one, two, three, or four registers

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1, A2, A3, and A4) and T32 (T1, T2, T3, and T4).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VLD1{<c>}{<q>}.<size> <list>, [(<Rn>{:<align>})], <Rm>

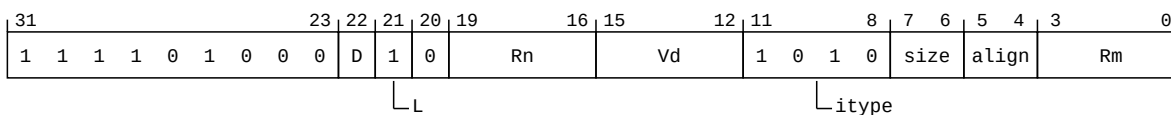
Decode for all variants of this encoding

```

if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```


A2



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [**<Rn>**{:<align>}]!

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D:Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

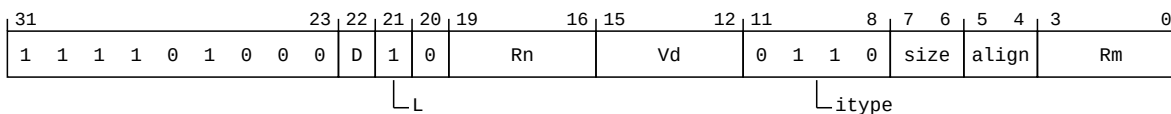
```

CONSTRAINED UNPREDICTABLE behavior

If $d + \text{regs} > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

$$\text{VLD1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

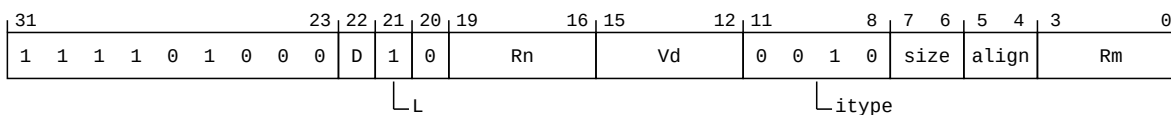
```

CONSTRAINED UNPREDICTABLE behavior

If $d_{\text{regs}} > 32$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A4



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

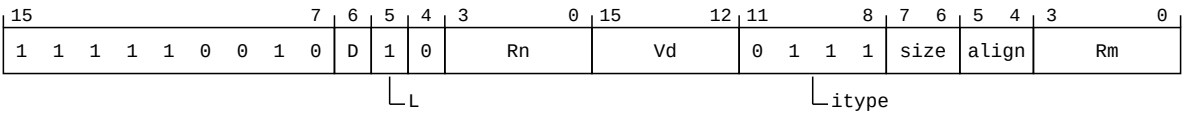
```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

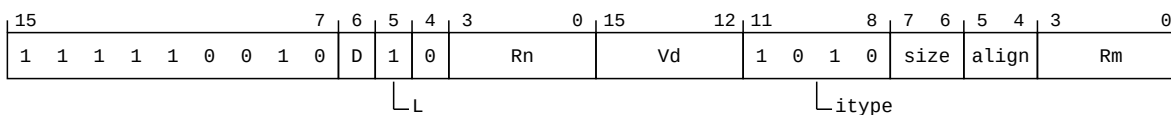
Decode for all variants of this encoding

```

if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D:Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

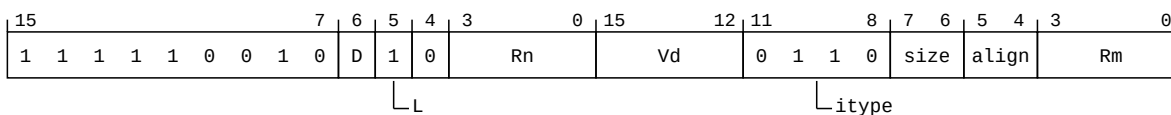
```

CONSTRAINED UNPREDICTABLE behavior

If $d + \text{regs} > 32$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

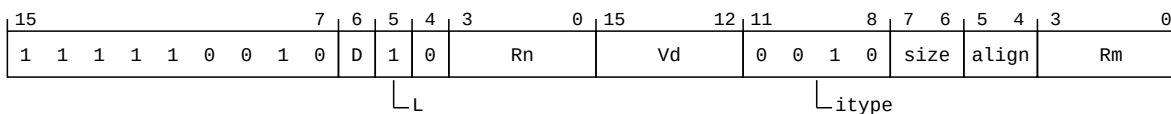
```
if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T4



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD1 \(multiple single elements\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, “A3 Post-indexed”, “A4 Offset”, and “A4 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, “T3 Post-indexed”, “T4 Offset”, and “T4 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	64

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- { <Dd> } Single register. Selects the A1 and T1 encodings of the instruction.
- { <Dd>, <Dd+1> } Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
- { <Dd>, <Dd+1>, <Dd+2> } Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
- { <Dd>, <Dd+1>, <Dd+2>, <Dd+3> } Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “align” field as 0b00.

Whenever <align> is present, the permitted values are:

- 64** 64-bit alignment, encoded in the “align” field as 0b01.
- 128** 128-bit alignment, encoded in the “align” field as 0b10. Available only if <list> contains two or four registers.
- 256** 256-bit alignment, encoded in the “align” field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
```

```

let privileged : boolean = PSTATE.EL != EL0;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
if !IsAlignedSize(address, alignment) then
    let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
    AArch32_Abort(fault);
end;

for r = 0 to regs-1 do
    for e = 0 to elements-1 do
        var data : bits(ebytes*8);
        if ebytes != 8 then
            data = MemU{8*ebytes}(address);
        else
            if !IsAlignedSize(address, ebytes) && AlignmentEnforced() then
                let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
                AArch32_Abort(fault);
            end;

            if BigEndian(AccessType_ASIMD) then
                data[31:0] = MemU{32}(address+4);
                data[63:32] = MemU{32}(address);
            else
                data[31:0] = MemU{32}(address);
                data[63:32] = MemU{32}(address+4);
            end;
        end;

        D(d+r)[e:(8*ebytes)] = data;
        address = address + ebytes;
    end;
end;

if wback then
    if register_index then
        R(n) = R(n) + R(m);
    else
        R(n) = R(n) + 8*regs;
    end;
end;
end;

```

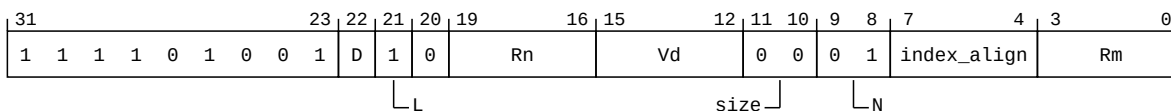
C6.1.87 VLD2 (single 2-element structure to one lane)

Load single 2-element structure to one lane of two registers loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#), [A2](#), and [A3](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;

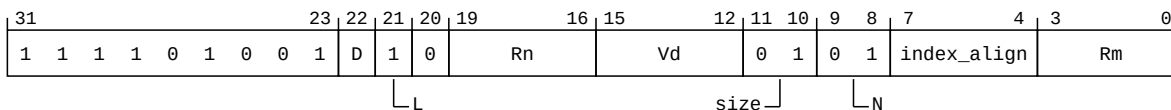
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

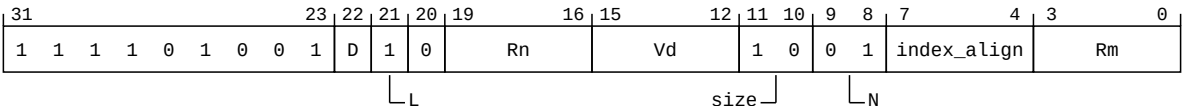
```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

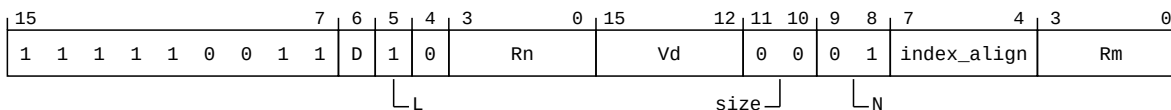
```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

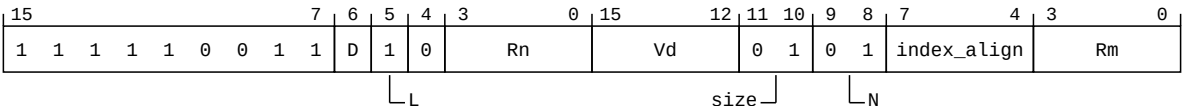
```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

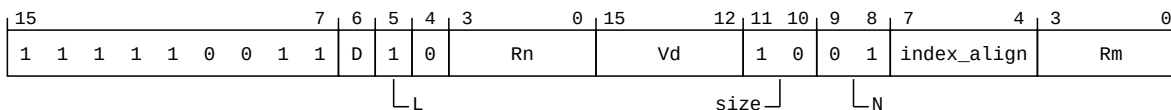
```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD2 (single 2-element structure to all lanes)"); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD2 \(single 2-element structure to one lane\)](#).

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>] } Single-spaced registers, encoded as “spacing” = 0.
{ <Dd>[<index>], <Dd+2>[<index>] } Double-spaced registers, encoded as “spacing” = 1. Not permitted when <size> = 8.

The encoding of “spacing” depends on <size>:

<size> = 16 “spacing” is encoded in the “index_align<1>” field.
<size> = 32 “spacing” is encoded in the “index_align<2>” field.

The register <Dd> is encoded in the “D:Vd” field.

The permitted values and encoding of <index> depend on <size>:

<size> = 8 <index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.
<size> = 16 <index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.
<size> = 32 <index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> = 8 Encoded in the “index_align<0>” field as 0.
<size> = 16 Encoded in the “index_align<0>” field as 0.
<size> = 32 Encoded in the “index_align<1:0>” field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

$\langle \text{size} \rangle == 8$ $\langle \text{align} \rangle$ is 16, meaning 16-bit alignment, encoded in the “index_align<0>” field as 1.
 $\langle \text{size} \rangle == 16$ $\langle \text{align} \rangle$ is 32, meaning 32-bit alignment, encoded in the “index_align<0>” field as 1.
 $\langle \text{size} \rangle == 32$ $\langle \text{align} \rangle$ is 64, meaning 64-bit alignment, encoded in the “index_align<1:0>” field as 0b01.

: is the preferred separator before the $\langle \text{align} \rangle$ value, but the alignment can be specified as @ $\langle \text{align} \rangle$, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    D(d)[index*:(8*bytes)] = MemU{8*bytes}(address);
    D(d2)[index*:(8*bytes)] = MemU{8*bytes}(address+bytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*bytes;
        end;
    end;
end;
end;

```

C6.1.88 VLD2 (single 2-element structure to all lanes)

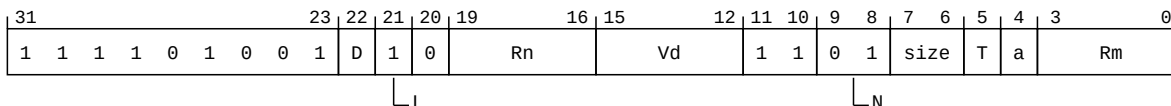
Load single 2-element structure and replicate to all lanes of two registers

Load single 2-element structure and replicate to all lanes of two registers loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VLD2{<c>}{<q>}.<size> <list>, [**<Rn>**{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let alignment : integer{} = if a == '0' then 1 else 2*ebytes;
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;

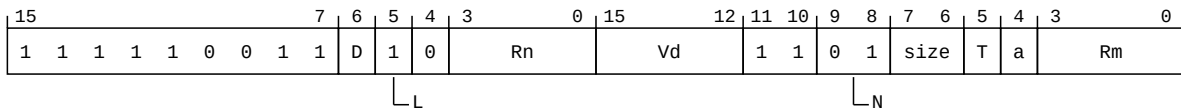
```

CONSTRAINED UNPREDICTABLE behavior

If $d_2 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VLD2}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let alignment : integer{} = if a == '0' then 1 else 2*ebytes;
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $d_2 > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly **VLD2** (single 2-element structure to all lanes).

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of two SIMD&FP registers.

The list must be one of:

{ <Dd>[], <Dd+1>[] } Single-spaced registers, encoded in the “T” field as 0.

{ <Dd>[], <Dd+2>[] } Double-spaced registers, encoded in the “T” field as 1.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “a” field as 0.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8 <align> is 16, meaning 16-bit alignment, encoded in the “a” field as 1.

<size> == 16 <align> is 32, meaning 32-bit alignment, encoded in the “a” field as 1.

<size> == 32 <align> is 64, meaning 64-bit alignment, encoded in the “a” field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    let esize : integer{} = 8 * ebytes;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*ebytes;
        end;
    end;
end;
```

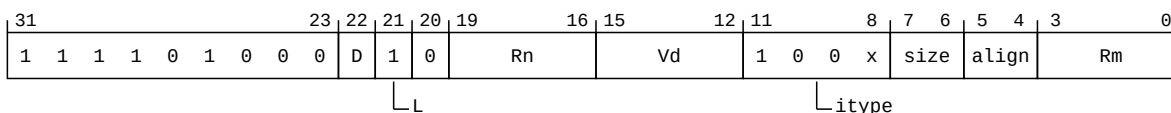
C6.1.89 VLD2 (multiple 2-element structures)

Load multiple 2-element structures to two or four registers loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see [Element and structure load/store instructions](#). Every element of each register is loaded. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;

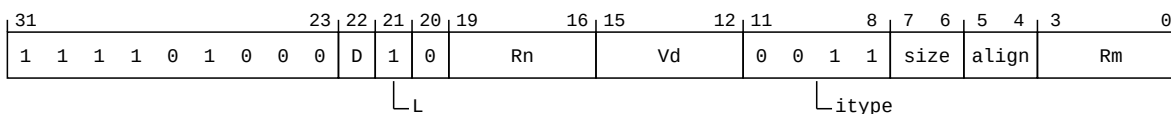
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

$$\text{VLD2}\{\langle c \rangle\}\{\langle g \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;

let pairs : integer = 2;

let inc : integer = 2;

let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);

let ebytes : integer{} = 1 << UInt(size);

let elements : integer = 8 DIV ebytes;

let d : integer = UInt(D::Vd);

let d2 : integer = d + inc;

let n : integer = UInt(Rn);

let m : integer = UInt(Rm);

let wback : boolean = (m != 15);

let register_index : boolean = (m != 15 && m != 13);

if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;

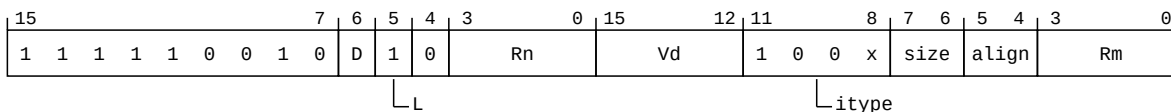
```

CONSTRAINED UNPREDICTABLE behavior

If $d_{2+\text{pairs}} > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

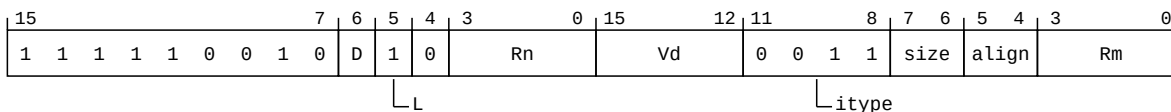
```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == 'l1' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD2 \(multiple 2-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, and “A2 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, and “T2 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- { <Dd>, <Dd+1> } Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as 0b1000.
- { <Dd>, <Dd+2> } Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as 0b1001.
- { <Dd>, <Dd+1>, <Dd+2>, <Dd+3> } Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “align” field as 0b00.

Whenever <align> is present, the permitted values are:

- 64** 64-bit alignment, encoded in the “align” field as 0b01.
- 128** 128-bit alignment, encoded in the “align” field as 0b10.
- 256** 256-bit alignment, encoded in the “align” field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to pairs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e:(8*ebytes)] = MemU{8*ebytes}(address);
            D(d2+r)[e:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
            address = address + 2*ebytes;
        end;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 16*pairs;
        end;
    end;
end;

```

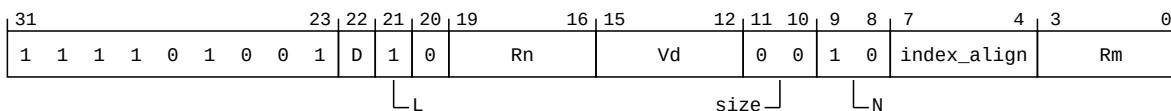
C6.1.90 VLD3 (single 3-element structure to one lane)

Load single 3-element structure to one lane of three registers loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#), [A2](#), and [A3](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

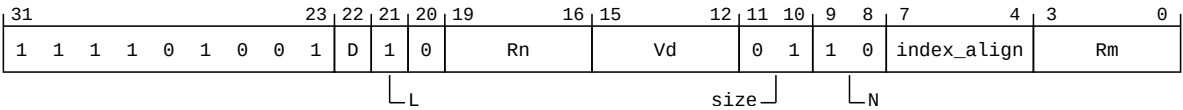
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

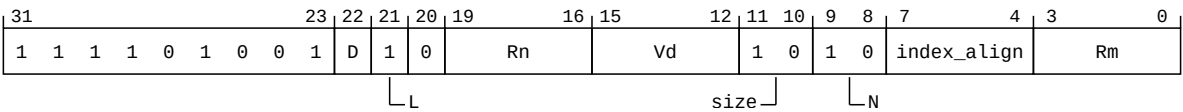
```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

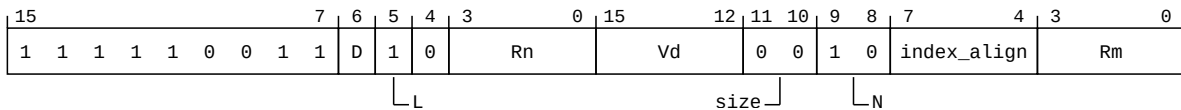
```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

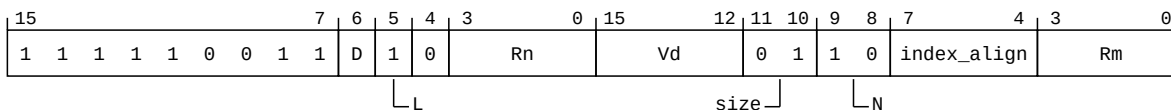
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

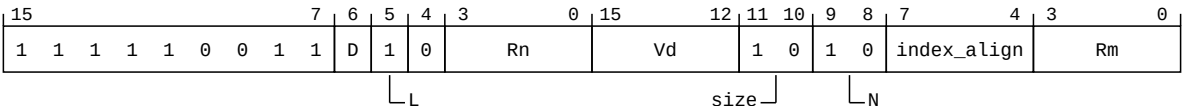
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD3 (single 3-element structure to all lanes)"); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD3 \(single 3-element structure to one lane\)](#).

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.

The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] } Single-spaced registers, encoded as “spacing” = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] } Double-spaced registers, encoded as “spacing” = 1. Not permitted when <size> == 8.

The encoding of “spacing” depends on <size>:

<size> == 8 “spacing” is encoded in the “index_align<0>” field.

<size> == 16 “spacing” is encoded in the “index_align<1>” field, and “index_align<0>” is set to 0.

<size> == 32 “spacing” is encoded in the “index_align<2>” field, and “index_align<1:0>” is set to 0b00.

The register <Dd> is encoded in the “D:Vd” field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8 <index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.

<size> == 16 <index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.

<size> == 32 <index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Alignment

Standard alignment rules apply, see Alignment support.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    D(d)[index*:(8*ebytes)] = MemU{8*ebytes}(address);
    D(d2)[index*:(8*ebytes)] = MemU{8*ebytes}(address+ebytes);
    D(d3)[index*:(8*ebytes)] = MemU{8*ebytes}(address+2*ebytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 3*ebytes;
        end;
    end;
end;
```

C6.1.91 VLD3 (single 3-element structure to all lanes)

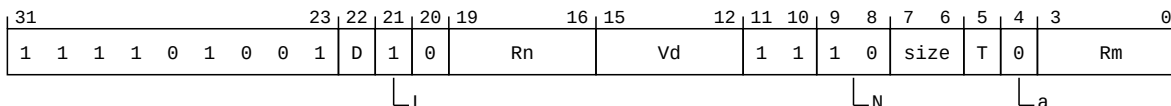
Load single 3-element structure and replicate to all lanes of three registers

Load single 3-element structure and replicate to all lanes of three registers loads one 3-element structure from memory into all lanes of three registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>} {<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' || a == '1' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

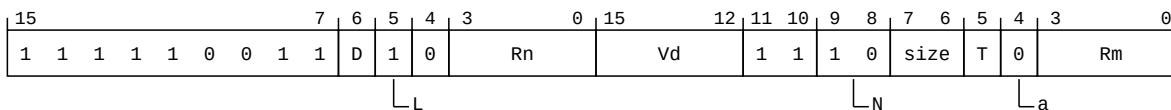
```

CONSTRAINED UNPREDICTABLE behavior

If $d_3 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' || a == '1' then Undefined(); end;
let ebytes : integer{} = 1 << UInt(size);
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $d_3 > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD3 \(single 3-element structure to all lanes\)](#).

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of three SIMD&FP registers.

The list must be one of:

{ <Dd>[], <Dd+1>[], <Dd+2>[] } Single-spaced registers, encoded in the “T” field as 0.

{ <Dd>[], <Dd+2>[], <Dd+4>[] } Double-spaced registers, encoded in the “T” field as 1.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Alignment

Standard alignment rules apply, see Alignment support.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    let esize : integer{} = ebytes * 8;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    let element3 : bits(esize) = MemU{8*ebytes}(address+2*ebytes);

    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);
    D(d3) = Replicate{64}(element3);
    if wback then
        if register_index then
```

```
        R(n) = R(n) + R(m);  
    else  
        R(n) = R(n) + 3*ebytes;  
    end;  
end;  
end;
```

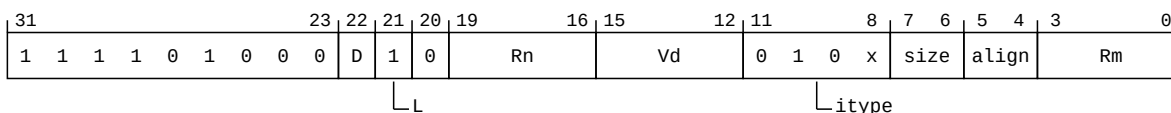
C6.1.92 VLD3 (multiple 3-element structures)

Load multiple 3-element structures to three registers loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see [Element and structure load/store instructions](#). Every element of each register is loaded. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if itype != '010x' then See("Related encodings"); end;
if size == '11' || align[1] == '1' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

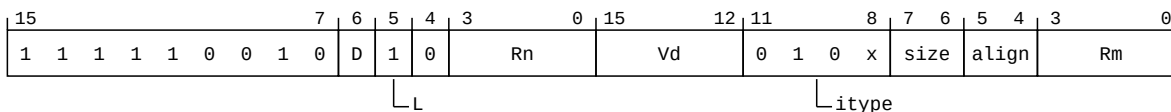
```

CONSTRAINED UNPREDICTABLE behavior

If $d_3 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if itype != '010x' then See("Related encodings"); end;
if size == '11' || align[1] == '1' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD3 \(multiple 3-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- { <Dd>, <Dd+1>, <Dd+2> } Single-spaced registers, encoded in the “itype” field as 0b0100.
- { <Dd>, <Dd+2>, <Dd+4> } Double-spaced registers, encoded in the “itype” field as 0b0101.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “align” field as 0b00.

Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the “align” field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about <Rn>, !, and <Rm>, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

var address : bits(32) = R(n);
```

```
let nontemporal : boolean = FALSE;
let privileged : boolean = PSTATE.EL != ELO;
let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
if !IsAlignedSize(address, alignment) then
    let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
    AArch32_Abort(fault);
end;

for e = 0 to elements-1 do
    D(d)[e:(8*bytes)] = MemU{8*bytes}(address);
    D(d2)[e:(8*bytes)] = MemU{8*bytes}(address+bytes);
    D(d3)[e:(8*bytes)] = MemU{8*bytes}(address+2*bytes);
    address = address + 3*bytes;
end;

if wback then
    if register_index then
        R(n) = R(n) + R(m);
    else
        R(n) = R(n) + 24;
    end;
end;
end;
```

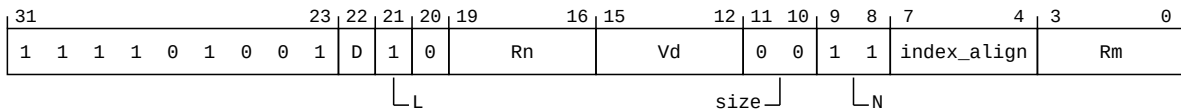
C6.1.93 VLD4 (single 4-element structure to one lane)

Load single 4-element structure to one lane of four registers loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#), [A2](#), and [A3](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

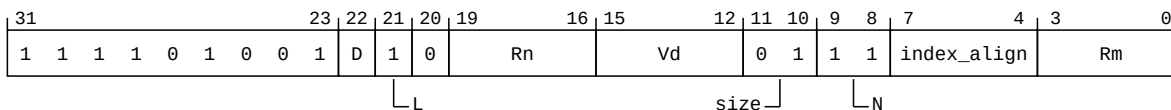
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

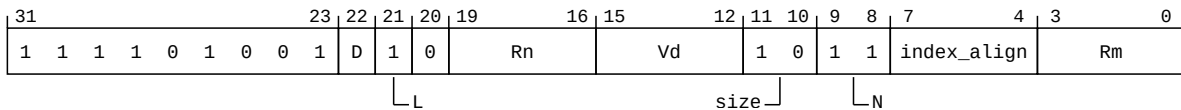
```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

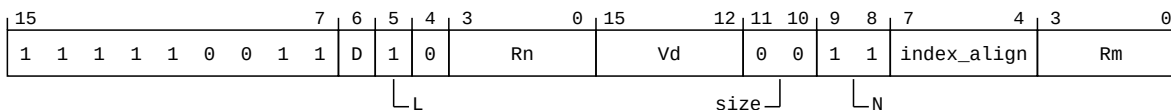
```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

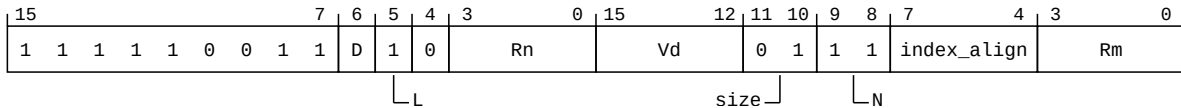
```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

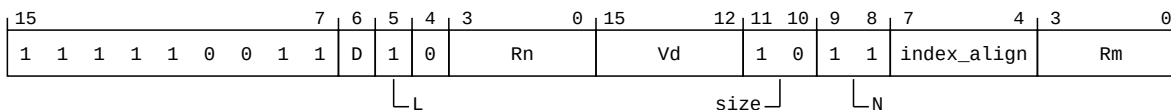
```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then See("VLD4 (single 4-element structure to all lanes)"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD4 \(single 4-element structure to one lane\)](#).

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.

The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] } Single-spaced registers, encoded as “spacing” = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] } Double-spaced registers, encoded as “spacing” = 1. Not permitted when <size> == 8.

The encoding of “spacing” depends on <size>:

<size> == 16 “spacing” is encoded in the “index_align<1>” field.

<size> == 32 “spacing” is encoded in the “index_align<2>” field.

The register <Dd> is encoded in the “D:Vd” field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8 <index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.

<size> == 16 <index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.

<size> == 32 <index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> == 8 Encoded in the “index_align<0>” field as 0.

<size> == 16 Encoded in the “index_align<0>” field as 0.

<size> == 32 Encoded in the “index_align<1:0>” field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

$\langle \text{size} \rangle == 8$ $\langle \text{align} \rangle$ is 32, meaning 32-bit alignment, encoded in the “index_align<0>” field as 1.
 $\langle \text{size} \rangle == 16$ $\langle \text{align} \rangle$ is 64, meaning 64-bit alignment, encoded in the “index_align<0>” field as 1.
 $\langle \text{size} \rangle == 32$ $\langle \text{align} \rangle$ can be 64 or 128. 64-bit alignment is encoded in the “index_align<1:0>” field as 0b01,
and 128-bit alignment is encoded in the “index_align<1:0>” field as 0b10.

: is the preferred separator before the $\langle \text{align} \rangle$ value, but the alignment can be specified as @ $\langle \text{align} \rangle$, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    D(d)[index*:(8*abytes)] = MemU{8*abytes}(address);
    D(d2)[index*:(8*abytes)] = MemU{8*abytes}(address+abytes);
    D(d3)[index*:(8*abytes)] = MemU{8*abytes}(address+2*abytes);
    D(d4)[index*:(8*abytes)] = MemU{8*abytes}(address+3*abytes);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*abytes;
        end;
    end;
end;
end;

```

C6.1.94 VLD4 (single 4-element structure to all lanes)

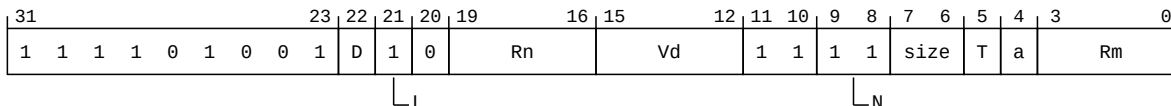
Load single 4-element structure and replicate to all lanes of four registers

Load single 4-element structure and replicate to all lanes of four registers loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [**<Rn>**:{<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' && a == '0' then Undefined(); end;
let ebytes : integer{} = if size == '11' then 4 else 1 << UInt(size);
let alignment : integer{} = (if a == '0' && size != '11' then 1
                             else 4 << (UInt(size[1]) + UInt(size[0])));
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

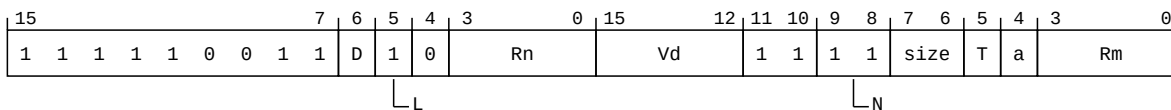
```

CONSTRAINED UNPREDICTABLE behavior

If $d_4 > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' && a == '0' then Undefined(); end;
let ebytes : integer{} = if size == '11' then 4 else 1 << UInt(size);
let alignment : integer{} = (if a == '0' && size != '11' then 1
                             else 4 << (UInt(size[1]) + UInt(size[0])));
let inc : integer = if T == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD4 \(single 4-element structure to all lanes\)](#).

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
1x	32

<list>

Is a list containing the 64-bit names of four SIMD&FP registers.

The list must be one of:

{ <Dd>[], <Dd+1>[], <Dd+2>[], <Dd+3>[] } Single-spaced registers, encoded in the “T” field as 0.

{ <Dd>[], <Dd+2>[], <Dd+4>[], <Dd+6>[] } Double-spaced registers, encoded in the “T” field as 1.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “a” field as 0.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8 <align> is 32, meaning 32-bit alignment, encoded in the “a” field as 1.

<size> == 16 <align> is 64, meaning 64-bit alignment, encoded in the “a” field as 1.

<size> == 32 <align> can be 64 or 128. 64-bit alignment is encoded in the “a:size<0>” field as 0b10, and 128-bit alignment is encoded in the “a:size<0>” field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    let esize : integer{} = ebytes * 8;
    let element1 : bits(esize) = MemU{8*ebytes}(address);
    let element2 : bits(esize) = MemU{8*ebytes}(address+ebytes);
    let element3 : bits(esize) = MemU{8*ebytes}(address+2*ebytes);
    let element4 : bits(esize) = MemU{8*ebytes}(address+3*ebytes);
    D(d) = Replicate{64}(element1);
    D(d2) = Replicate{64}(element2);
    D(d3) = Replicate{64}(element3);
    D(d4) = Replicate{64}(element4);
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*ebytes;
        end;
    end;
end;
end;

```

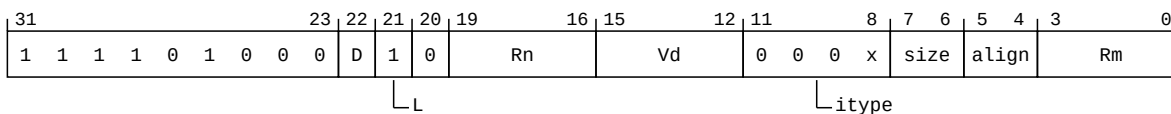
C6.1.95 VLD4 (multiple 4-element structures)

Load multiple 4-element structures to four registers loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see [Element and structure load/store instructions](#). Every element of each register is loaded. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if itype != '000x' then See("Related encodings"); end;
if size == '11' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

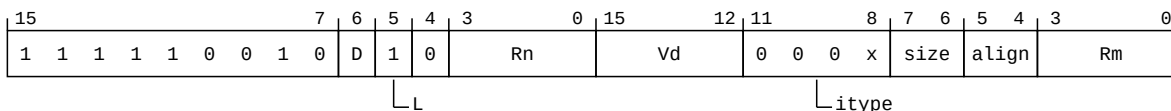
```

CONSTRAINED UNPREDICTABLE behavior

If $d4 > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Offset variant

Applies when $(Rm == 1111)$

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when $(Rm == 1101)$

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when $(Rm != 11x1)$

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if itype != '000x' then See("Related encodings"); end;
if size == '11' then Undefined(); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLD4 \(multiple 4-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- { `<Dd>`, `<Dd+1>`, `<Dd+2>`, `<Dd+3>` } Single-spaced registers, encoded in the “itype” field as `0b0000`.
- { `<Dd>`, `<Dd+2>`, `<Dd+4>`, `<Dd+6>` } Double-spaced registers, encoded in the “itype” field as `0b0001`.

The register `<Dd>` is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “align” field as `0b00`.

Whenever `<align>` is present, the permitted values are:

- 64** 64-bit alignment, encoded in the “align” field as 0b01.
- 128** 128-bit alignment, encoded in the “align” field as 0b10.
- 256** 256-bit alignment, encoded in the “align” field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for e = 0 to elements-1 do
        D(d)[e:(8*bytes)] = MemU{8*bytes}(address);
        D(d2)[e:(8*bytes)] = MemU{8*bytes}(address+bytes);
        D(d3)[e:(8*bytes)] = MemU{8*bytes}(address+2*bytes);
        D(d4)[e:(8*bytes)] = MemU{8*bytes}(address+3*bytes);
        address = address + 4*bytes;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 32;
        end;
    end;
end;

```

C6.1.96 VLDM, VLDMDB, VLDMIA

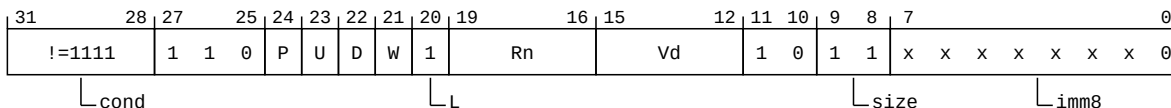
Load Multiple SIMD&FP registers loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the alias [VPOP](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

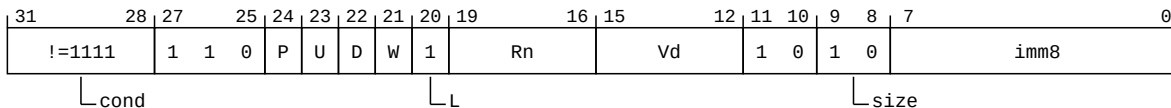
If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If regs > 16 || (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> // (Preferred syntax)
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

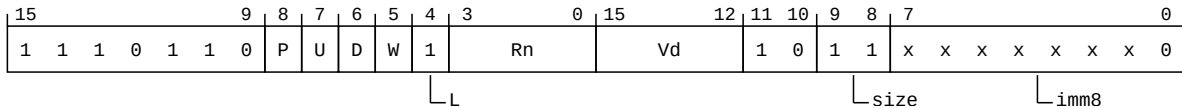
If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

If (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist> // (Preferred syntax)
```

```
VLDmia{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

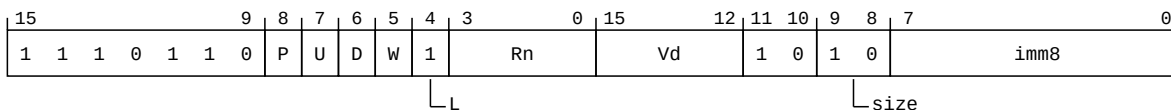
If `reqs == 0`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction operates as a **VLDM** with the same addressing mode but loads no registers.

If $\text{regs} > 16 \ || \ (\text{d+regs}) > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. If the instruction specifies writeback, the base register becomes `UNKNOWN`. This behavior does not affect any general-purpose registers.

T2



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> // (Preferred syntax)
```

```
VLDmia{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VLDR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd:D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (dregs) > 32 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction operates as a **VLDM** with the same addressing mode but loads no registers.

If $(d + \text{regs}) > 32$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- One or more of the SIMD and floating-point registers are **UNKNOWN**. If the instruction specifies writeback, the base register becomes **UNKNOWN**. This behavior does not affect any general-purpose registers.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VLDM](#).

Related encodings: See [Advanced SIMD and floating-point 64-bit move](#) for the T32 instruction set, or [Advanced SIMD and floating-point 64-bit move](#) for the A32 instruction set.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If writeback is not specified, the PC can be used.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

!

Specifies base register writeback, encoded in “W”:

W	!
0	[absent]
1	[present]

<sreglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “Vd:D”, and “imm8” is set to the number of registers in the list. The list must contain at least one register.

Alias Conditions

Alias	Is preferred when
VPOP	P == '0' && U == '1' && W == '1' && Rn == '1101'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;

    for r = 0 to regs-1 do
        if single_regs then
            S(d+r) = MemA{32}(address);
            address = address+4;
        else
```

```
    let word1 : bits(32) = MemA{32}(address);
    let word2 : bits(32) = MemA{32}(address+4);
    address = address+8;

    // Combine the word-aligned words in the correct order for current endianness.
    D(d+r) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
end;

end;

if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

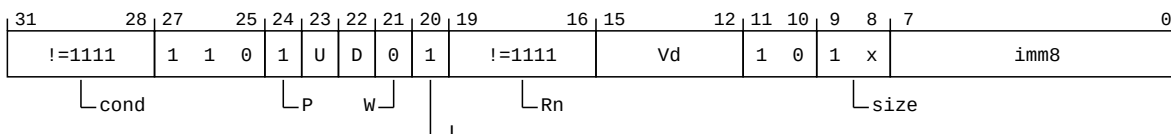
C6.1.97 VLDR (immediate)

Load SIMD&FP register (immediate) loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #(+/-)<imm>}]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #(+/-)<imm>}]
```

Decode for all variants of this encoding

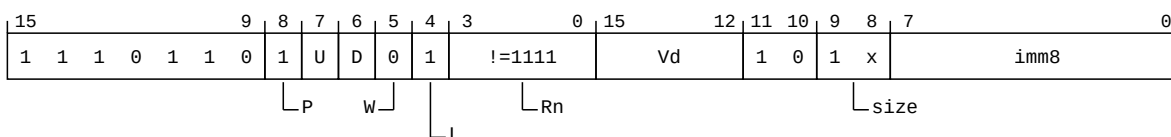
```
if size != '1x' then Undefined(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #(+/-)<imm>}]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #(+/-)<imm>}]
```

Decode for all variants of this encoding

```
if size != 'lx' then Undefined(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

.32

Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	–
1	+

<imm>

is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the “imm8” field as <imm>/4.

.64

Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    case esize of
        when 32 =>
            S(d) = MemA{32}(address);
        when 64 =>
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            // Combine the word-aligned words in the correct order for current endianness.
            D(d) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
    end;
end;
```

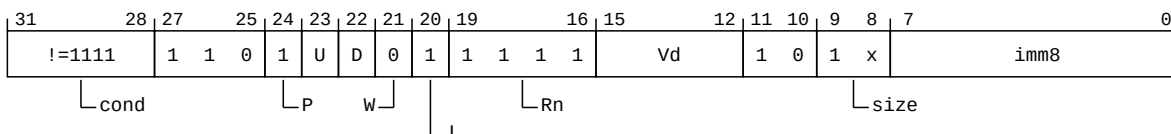
C6.1.98 VLDR (literal)

Load SIMD&FP register (literal) loads a single register from the Advanced SIMD and floating-point register file, using an address from the PC value and an immediate offset.

Depending on settings in the **CPACR**, **NSACR**, **HCPTR**, and **FPEXC** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}{.32} <Sd>, [PC, # {+/-}<imm>]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}{.64} <Dd>, [PC, # {+/-}<imm>]
```

Decode for all variants of this encoding

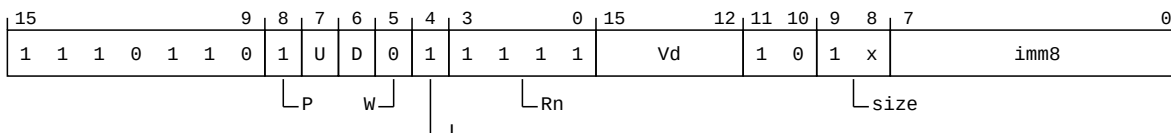
```
if size != '1x' then Undefined(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}{.32} <Sd>, [PC, # {+/-}<imm>]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, <label> // (Preferred syntax)
```

```
VLDR{<c>}{<q>}{.64} <Dd>, [PC, # {+/-}<imm>]
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as **NOP**. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

.32

Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<label>

The label of the literal data item to be loaded.

For the single-precision scalar or double-precision scalar variants: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020.

For the half-precision scalar variant: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 2 in the range -510 to 510.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the “imm8” field as <imm>/4.

.64

Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see [Use of labels in UAL instruction syntax](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let base : bits(32) = if n == 15 then AlignDownSize(PC32(),4) else R(n);
    let address : bits(32) = if add then (base + imm32) else (base - imm32);
    case esize of
        when 32 =>
            S(d) = MemA{32}(address);
        when 64 =>
            let word1 : bits(32) = MemA{32}(address);
            let word2 : bits(32) = MemA{32}(address+4);
            // Combine the word-aligned words in the correct order for current endianness.
            D(d) = if BigEndian(AccessType_ASIMD) then word1::word2 else word2::word1;
    end;
end;
```

C6.1.99 VMAX (floating-point)

Vector Maximum (floating-point)

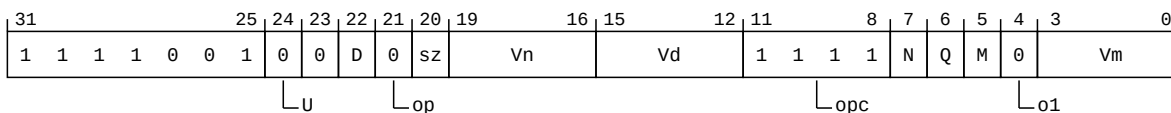
Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

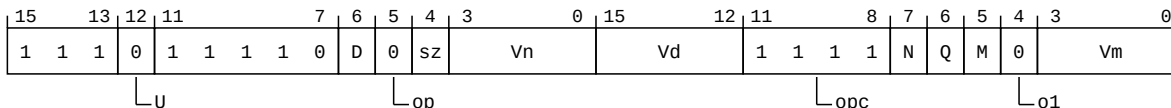
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, <Qn>, <Qm>}

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32; let elements : integer{} = 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Floating-point maximum and minimum

- $\max(+0.0, -0.0) = +0.0$
- If any input is a NaN, the corresponding result element is the default NaN.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = D(n+r)[e*:esize];
            let op2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMax{esize}(op1, op2, fpcr);
            else
                D(d+r)[e*:esize] = FPMin{esize}(op1, op2, fpcr);
            end;
        end;
    end;
end;
end;
```

C6.1.100 VMAX (integer)

Vector Maximum (integer)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

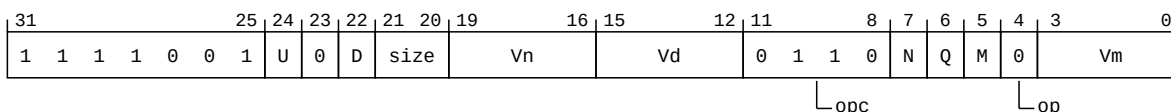
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

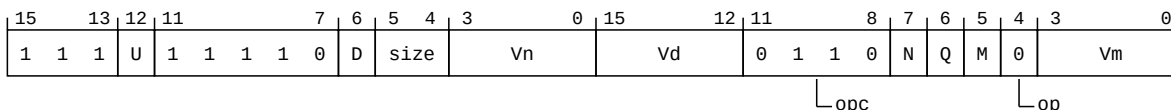
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if maximum then Max(element1, element2)
                                   else Min(element1, element2));
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

C6.1.101 **VMAXNM**

Floating-point Maximum Number

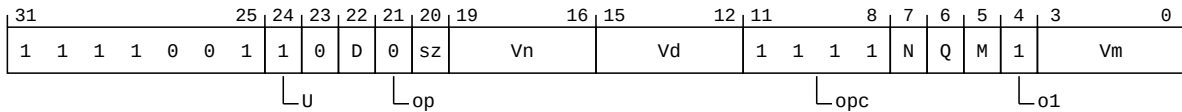
This instruction determines the floating-point maximum number.

It handles NaNs in consistence with the IEEE 754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point `VMAX`.

This instruction is not conditional.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VMAXNM{ <q> } .<dt> <Dd> , <Dn> , <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

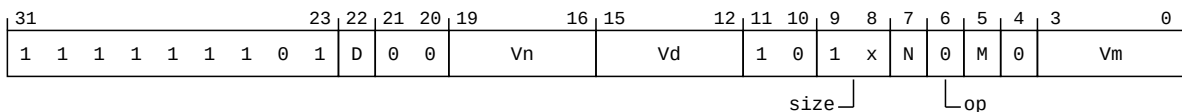
$$\text{VMAXNM}\{\langle q \rangle\}.\langle dt \rangle \quad \langle Qd \rangle, \langle Qn \rangle, \langle Qm \rangle$$
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Double-precision scalar variant

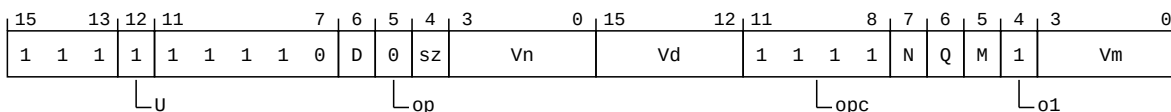
Applies when (size == 11)

```
VMAXNM(<q>).F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMAXNM(<q>).<dt> <Dd>, <Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMAXNM(<q>).<dt> <Qd>, <Qn>, <Qm>
```

Decode for all variants of this encoding

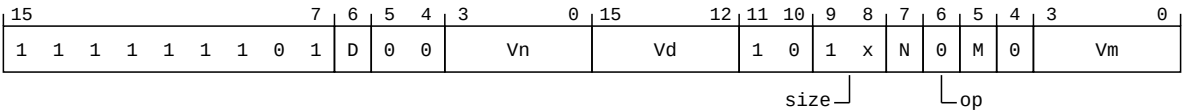
```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32;
let elements : integer = 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMAXNM(<q>).F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMAXNM(<q>).F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32

sz	<dt>
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
if advsimd then // Advanced SIMD instruction
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(n+r)[e*:esize];
            let operand2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMaxNum{esize}(operand1, operand2, fpcr);
            else
                D(d+r)[e*:esize] = FPMinNum{esize}(operand1, operand2, fpcr);
            end;
        end;
    end;
else // VFP instruction
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 =>
            if maximum then
                S(d) = FPMaxNum{32}(S(n), S(m), fpcr);
            else
                S(d) = FPMinNum{32}(S(n), S(m), fpcr);
```



```
        end;  
    when 64 =>  
        if maximum then  
            D(d) = FPMaxNum{64}(D(n), D(m), fpcr);  
        else  
            D(d) = FPMinNum{64}(D(n), D(m), fpcr);  
        end;  
    end;  
end;  
end;
```

C6.1.102 VMIN (floating-point)

Vector Minimum (floating-point)

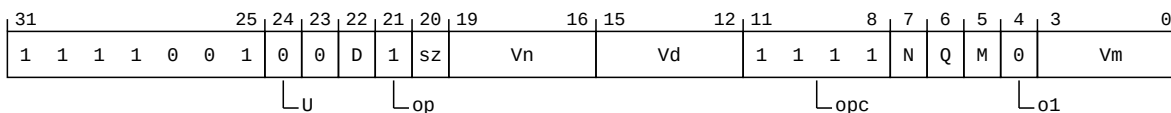
Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

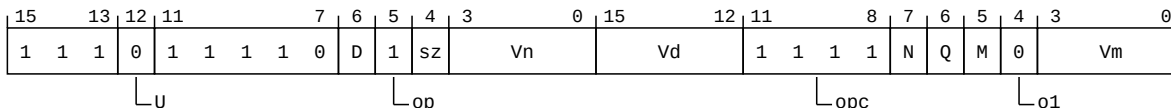
Applies when (Q == 1)

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, <Qn>, <Qm>}

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32; let elements : integer{} = 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If sz == '1' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Floating-point minimum

- $\min(+0.0, -0.0) = -0.0$
- If any input is a NaN, the corresponding result element is the default NaN.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : bits(esize) = D(n+r)[e*:esize];
            let op2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMax{esize}(op1, op2, fpcr);
            else
                D(d+r)[e*:esize] = FPMin{esize}(op1, op2, fpcr);
            end;
        end;
    end;
end;
end;
```

C6.1.103 VMIN (integer)

Vector Minimum (integer)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

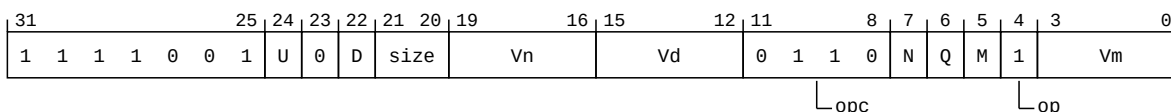
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

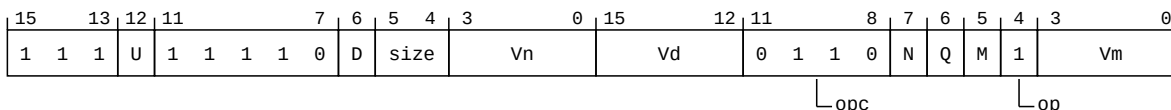
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (if maximum then Max(element1, element2)
                                   else Min(element1, element2));
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

C6.1.104 VMINNM

Floating-point Minimum Number

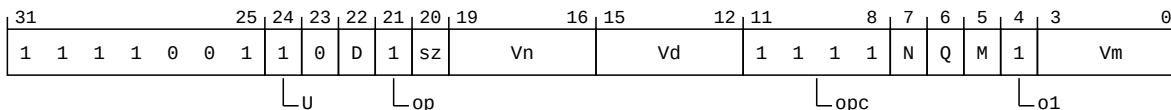
This instruction determines the floating point minimum number.

It handles NaNs in consistence with the IEEE 754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point `VMIN`.

This instruction is not conditional.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (`Q == 0`)

`VMINNM(<q>).<dt> <Dd>, <Dn>, <Dm>`

Encoding for the 128-bit SIMD vector variant

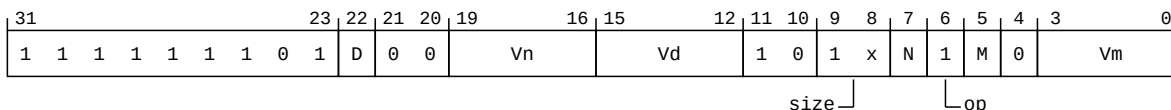
Applies when (`Q == 1`)

`VMINNM(<q>).<dt> <Qd>, <Qn>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2



Encoding for the Single-precision scalar variant

Applies when (`size == 10`)

`VMINNM(<q>).F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)`

Encoding for the Double-precision scalar variant

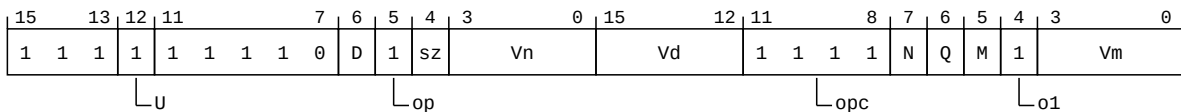
Applies when (size == 11)

VMINNM(<q>).F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMINNM(<q>).<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMINNM(<q>).<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

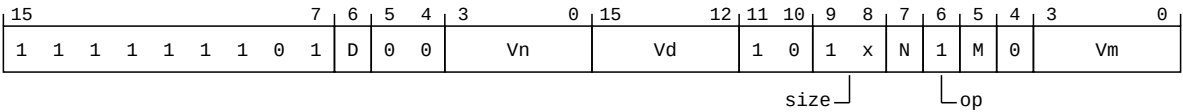
```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let advsimd : boolean = TRUE;
let esize : integer{} = 32;
let elements : integer = 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMINNM(<q>).F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMINNM(<q>).F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let maximum : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors.

Assembler Symbols

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32

sz	<dt>
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
if advsimd then // Advanced SIMD instruction
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let operand1 : bits(esize) = D(n+r)[e*:esize];
            let operand2 : bits(esize) = D(m+r)[e*:esize];
            if maximum then
                D(d+r)[e*:esize] = FPMaxNum{esize}(operand1, operand2, fpcr);
            else
                D(d+r)[e*:esize] = FPMinNum{esize}(operand1, operand2, fpcr);
            end;
        end;
    end;
else // VFP instruction
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 =>
            if maximum then
                S(d) = FPMaxNum{32}(S(n), S(m), fpcr);
            else
                S(d) = FPMinNum{32}(S(n), S(m), fpcr);
            end;
    end;
```

```
        end;  
    when 64 =>  
        if maximum then  
            D(d) = FPMaxNum{64}(D(n), D(m), fpcr);  
        else  
            D(d) = FPMinNum{64}(D(n), D(m), fpcr);  
        end;  
    end;  
end;  
end;
```

C6.1.105 VMLA (floating-point)

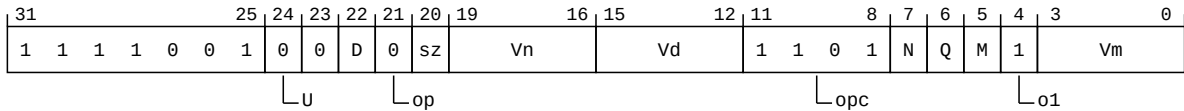
Vector Multiply Accumulate (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

`VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>`

Encoding for the 128-bit SIMD vector variant

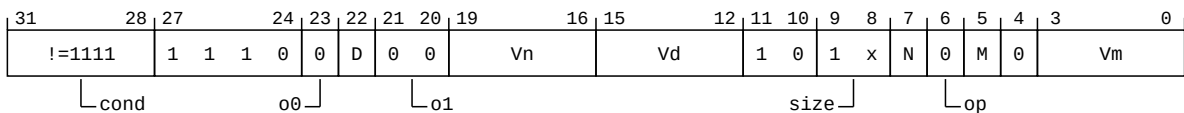
Applies when ($Q == 1$)

`VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>`

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2



Encoding for the Single-precision scalar variant

Applies when ($size == 10$)

`VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

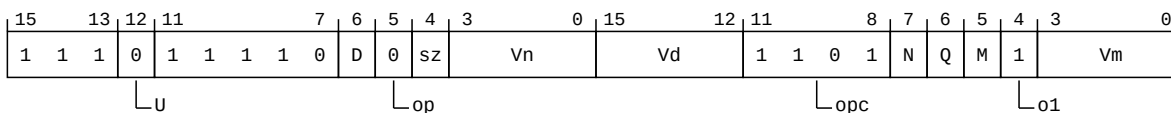
```
if size != '1x' then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

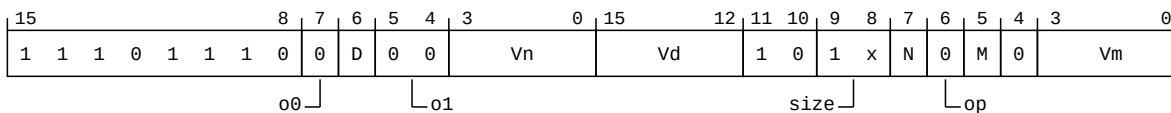
```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

`VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>`

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

`VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>`

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var addend : bits(esize) = FPMul{}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
                if !add then addend = FPNeg{esize}(addend, fpcr); end;
                D(d+r)[e*:esize] = FPAdd{esize}(D(d+r)[e*:esize], addend, fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 32 =>
                var addend32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
                if !add then addend32 = FPNeg{32}(addend32, fpcr); end;
                S(d) = FPAdd{32}(S(d), addend32, fpcr);
            when 64 =>
                var addend64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
                if !add then addend64 = FPNeg{64}(addend64, fpcr); end;
                D(d) = FPAdd{64}(D(d), addend64, fpcr);
        end;
    end;
end;
end;

```

C6.1.106 VMLA (integer)

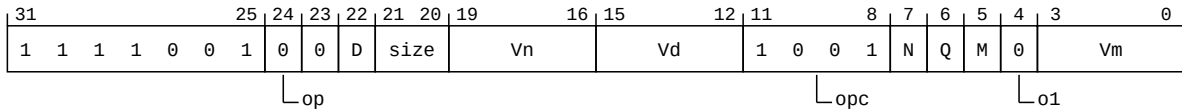
Vector Multiply Accumulate (integer)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and adds the products to the corresponding elements of the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

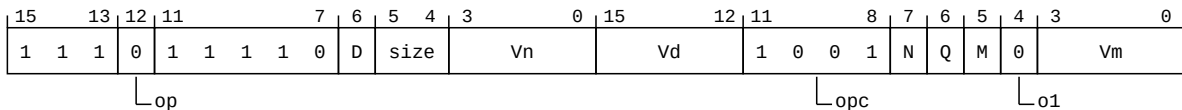
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;
```

C6.1.107 VMLA (by scalar)

Vector Multiply Accumulate (by scalar)

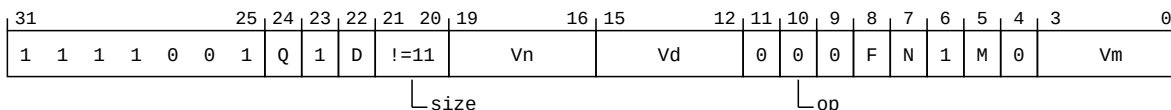
Vector Multiply Accumulate multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

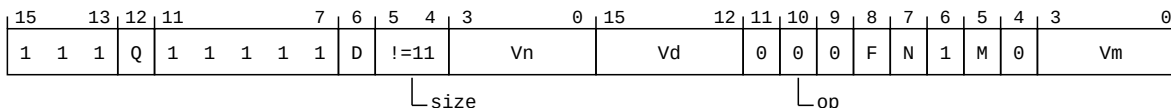
Decode for all variants of this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

F	size	<dt>
0	01	I16

F	size	<dt>
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in “Vm<2:0>”, and x is encoded in “M:Vm<3>”. If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in “Vm”, and x is encoded in “M”.

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                    fpcr),
                                                                    fpcr));
                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
end;
end;

```

C6.1.108 VMLAL (integer)

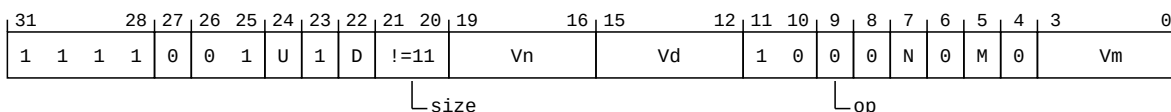
Vector Multiply Accumulate Long (integer)

Vector Multiply Accumulate Long multiplies corresponding elements in two vectors, and add the products to the corresponding element of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

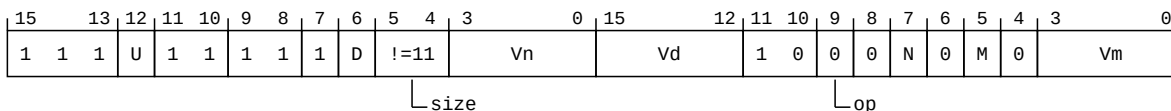
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;

```

T1



Encoding

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
```

```
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = Din(n+r)[e*:esize];
        let op2elt : bits(esize) = Din(m+r)[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let product : integer = element1 * element2;
        let addend : integer = if add then product else -product;
        if long_destination then
            Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
        else
            D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
        end;
    end;
end;
end;
```

C6.1.109 VMLAL (by scalar)

Vector Multiply Accumulate Long (by scalar)

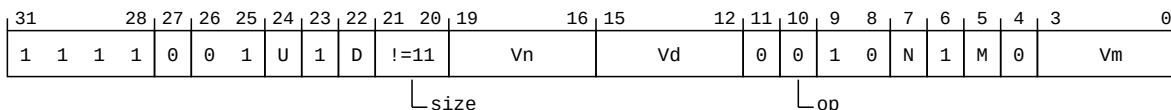
Vector Multiply Accumulate Long multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x>

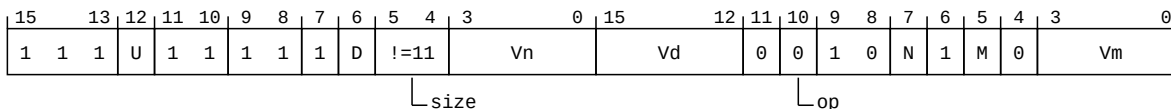
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);

```

T1



Encoding

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in “Vm<2:0>”, and x is encoded in “M:Vm<3>”. If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in “Vm”, and x is encoded in “M”.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                   fpcr),
                                                                                   fpcr));

                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
end;

```

C6.1.110 VMLS (floating-point)

Vector Multiply Subtract (floating-point)

Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

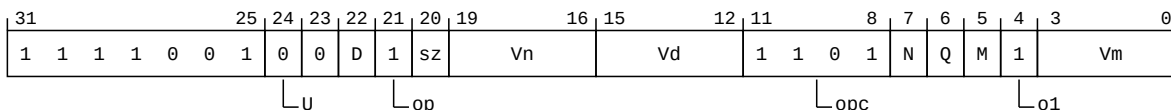
Note

Arm recommends that software does not use the VMLS instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

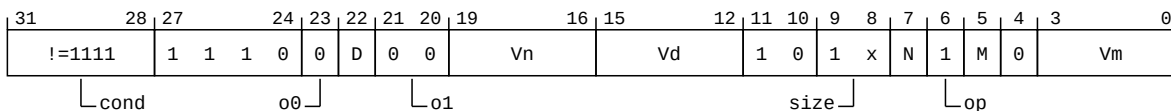
Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

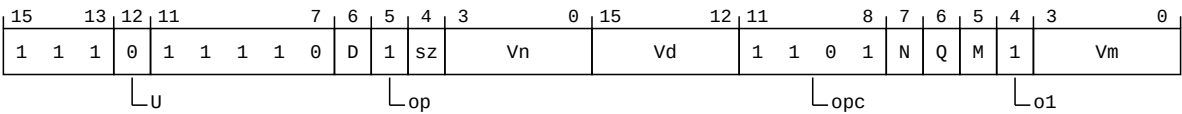
```
if size != '1x' then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY;
let regs : integer = ARBITRARY;
let elements : integer = ARBITRARY;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

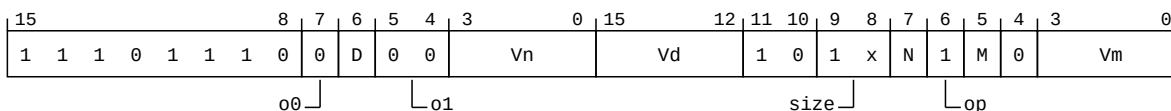
```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let add : boolean = (op == '0');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let add : boolean = (op == '0');
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                var addend : bits(esize) = FPMul{}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
                if !add then addend = FPNeg{esize}(addend, fpcr); end;
                D(d+r)[e*:esize] = FPAdd{esize}(D(d+r)[e*:esize], addend, fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 32 =>
                var addend32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
                if !add then addend32 = FPNeg{32}(addend32, fpcr); end;
                S(d) = FPAdd{32}(S(d), addend32, fpcr);
            when 64 =>
                var addend64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
                if !add then addend64 = FPNeg{64}(addend64, fpcr); end;
                D(d) = FPAdd{64}(D(d), addend64, fpcr);
        end;
    end;
end;
end;

```

C6.1.111 VMLS (integer)

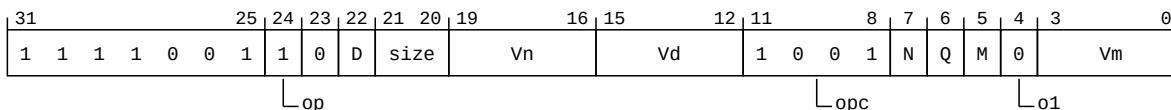
Vector Multiply Subtract (integer)

Vector Multiply Subtract multiplies corresponding elements in two vectors, and subtracts the products from the corresponding elements of the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

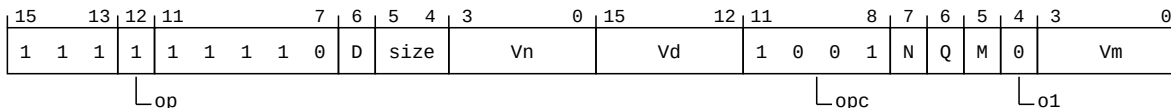
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = FALSE;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let product : integer = element1 * element2;
            let addend : integer = if add then product else -product;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
            else
                D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
            end;
        end;
    end;
end;
```

C6.1.112 VMLS (by scalar)

Vector Multiply Subtract (by scalar)

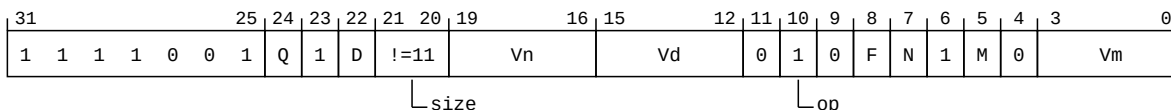
Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

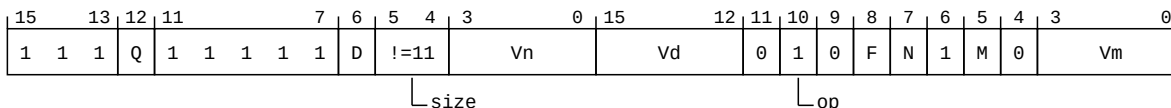
Decode for all variants of this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let add : boolean = (op == '0');
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

F	size	<dt>
0	01	I16

F	size	<dt>
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in “Vm<2:0>”, and x is encoded in “M:Vm<3>”. If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in “Vm”, and x is encoded in “M”.

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                   fpcr),
                                                                                   fpcr));
                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
end;

```


C6.1.113 VMLSL (integer)

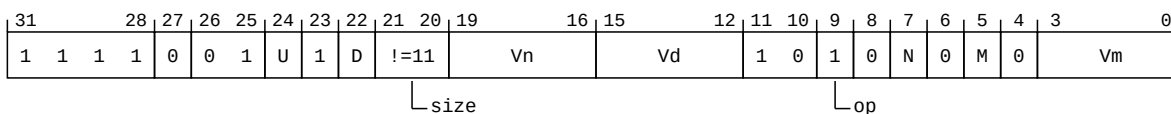
Vector Multiply Subtract Long (integer)

Vector Multiply Subtract Long multiplies corresponding elements in two vectors, and subtract the products from the corresponding elements of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

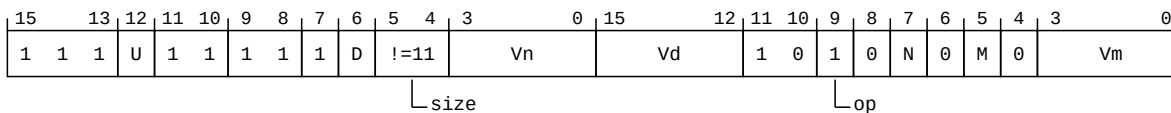
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;

```

T1



Encoding

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let long_destination : boolean = TRUE;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = 1;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
```

```
    for e = 0 to elements-1 do
        let op1elt : bits(esize) = Din(n+r)[e*:esize];
        let op2elt : bits(esize) = Din(m+r)[e*:esize];
        let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let product : integer = element1 * element2;
        let addend : integer = if add then product else -product;
        if long_destination then
            Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
        else
            D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
        end;
    end;
end;
```

C6.1.114 VMLSL (by scalar)

Vector Multiply Subtract Long (by scalar)

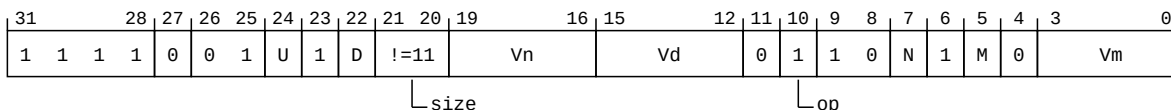
Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

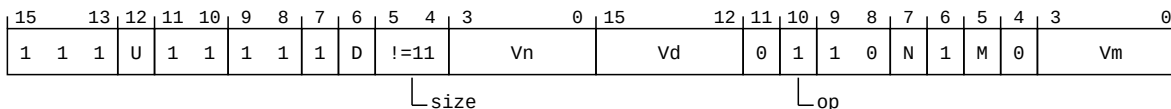
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);

```

T1



Encoding

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let add : boolean = (op == '0');
let floating_point : boolean = FALSE;
let long_destination : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in “Vm<2:0>”, and x is encoded in “M:Vm<3>”. If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in “Vm”, and x is encoded in “M”.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                let fp_addend : bits(esize) = (if add then FPMul{esize}(op1elt, op2elt, fpcr)
                                                else FPNeg{esize}(FPMul{esize}(op1elt, op2elt,
                                                                                   fpcr),
                                                                                   fpcr));

                D(d+r)[e*:esize] = FPAdd{esize}(Din(d+r)[e*:esize], fp_addend, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                let addend : integer = if add then element1*element2 else -element1*element2;
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = Qin(d>>1)[e*:(2*esize)] + addend;
                else
                    D(d+r)[e*:esize] = Din(d+r)[e*:esize] + addend;
                end;
            end;
        end;
    end;
end;
end;

```

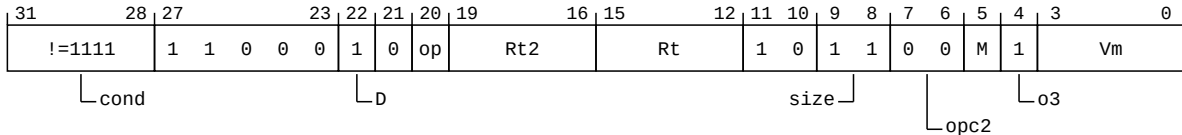
C6.1.115 VMOV (between two general-purpose registers and a doubleword floating-point register)

Copy two general-purpose registers to or from a SIMD&FP register copies two words from two general-purpose registers into a doubleword register in the Advanced SIMD and floating-point register file, or from a doubleword register in the Advanced SIMD and floating-point register file to two general-purpose registers.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the From general-purpose registers variant

Applies when (op == 0)

VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>

Encoding for the To general-purpose registers variant

Applies when (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>

Decode for all variants of this encoding

```

let to_arm_registers : boolean = (op == '1');
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(M::Vm);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;

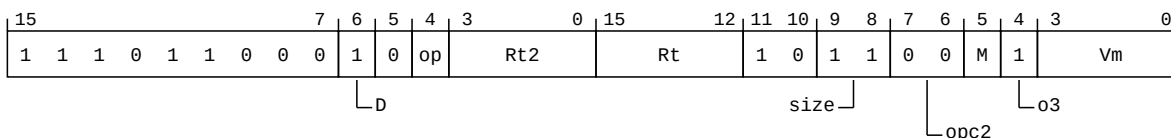
```

CONSTRAINED UNPREDICTABLE behavior

If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



Encoding for the From general-purpose registers variant

Applies when (op == 0)

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

Encoding for the To general-purpose registers variant

Applies when (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

Decode for all variants of this encoding

```
let to_arm_registers : boolean = (op == '1');
let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(M::Vm);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VMOV \(between two general-purpose registers and a doubleword floating-point register\)](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dm>

Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “M:Vm” field.

<Rt>

Is the first general-purpose register that <Dm>[31:0] will be transferred to or from, encoded in the “Rt” field.

<Rt2>

Is the second general-purpose register that <Dm>[63:32] will be transferred to or from, encoded in the “Rt2” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    if to_arm_registers then
        R(t) = D(m)[31:0];
        R(t2) = D(m)[63:32];
    else
        D(m)[31:0] = R(t);
        D(m)[63:32] = R(t2);
    end;
end;
```

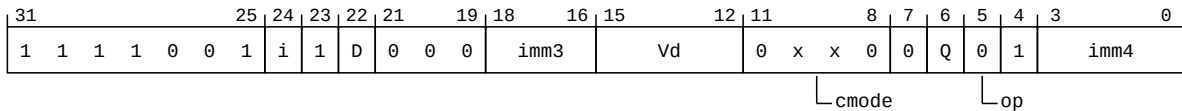
C6.1.116 VMOV (immediate)

Copy immediate value to a SIMD&FP register places an immediate constant into every element of the destination register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1, A2, A3, A4, and A5) and T32 (T1, T2, T3, T4, and T5).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

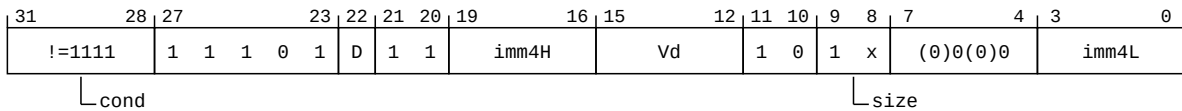
Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

Decode for all variants of this encoding

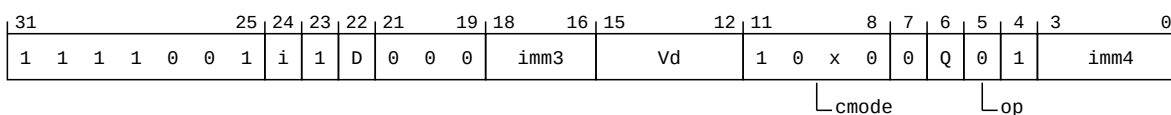
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let imm : bits(esize) = VFPEExpandImm{}(imm4H::imm4L);
let advsimd : boolean = FALSE;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let regs : integer = if size == '11' then 1 else 0;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A3



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

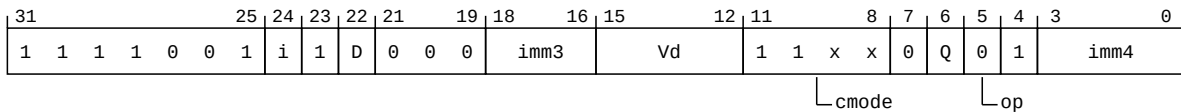
Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer() = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A4



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

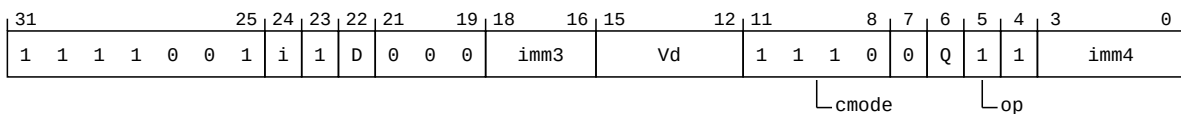
Applies when (Q == 1)

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A5



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I64 <Qd>, #<imm>
```

Decode for all variants of this encoding

```

if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;

```

T1

15	13	12	11	7	6	5	3	2	0	15	12	11	8	7	6	5	4	3	0				
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3	Vd	0	x	x	0	0	Q	0	1	imm4
															└ cmode			└ op					

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```

if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;

```

T2

1576543015121110987430																							
1	1	1	0	1	1	1	0	1	D	1	1	imm4H		Vd		1	0	1	x	(0)0(0)0		imm4L	

size

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

Decode for all variants of this encoding

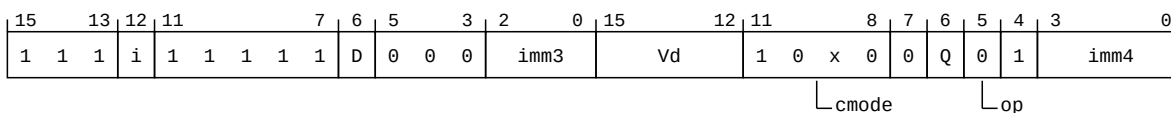
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let imm : bits(esize) = VFPEExpandImm{}(imm4H::imm4L);
let advsimd : boolean = FALSE;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let regs : integer = if size == '11' then 1 else 0;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T3



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

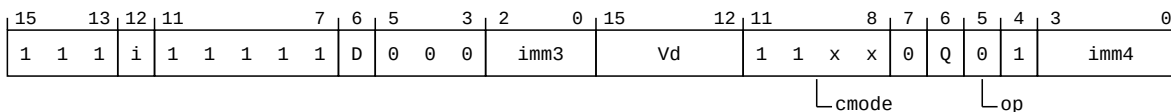
Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer() = 64;
let imm : bits(esize) = AdvSIMDEExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T4



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

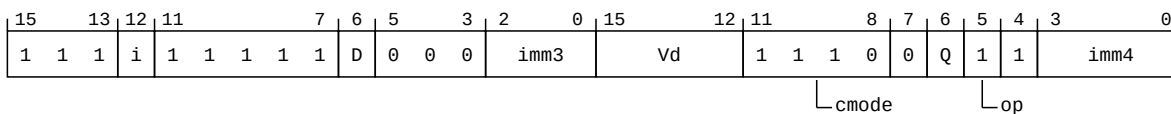
Applies when (Q == 1)

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T5



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMOV{<c>}{<q>}.I64 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if op == '0' && cmode[0] == '1' && cmode[3:2] != '11' then See("VORR (immediate)"); end;
if op == '1' && cmode != '1110' then See("Related encodings"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 64;
let imm : bits(esize) = AdvSIMDExpandImm(op, cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A3 128-bit SIMD vector”, “A3 64-bit SIMD vector”, “A4 128-bit SIMD vector”, “A4 64-bit SIMD vector”, “A5 128-bit SIMD vector”, and “A5 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, “T2 Single-precision scalar”, “T3 128-bit SIMD vector”, “T3 64-bit SIMD vector”, “T4 128-bit SIMD vector”, “T4 64-bit SIMD vector”, “T5 128-bit SIMD vector”, and “T5 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A3 128-bit SIMD vector”, “A3 64-bit SIMD vector”, “A4 128-bit SIMD vector”, “A4 64-bit SIMD vector”, “A5 128-bit SIMD vector”, “A5 64-bit SIMD vector”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T3 128-bit SIMD vector”, “T3 64-bit SIMD vector”, “T4 128-bit SIMD vector”, “T4 64-bit SIMD vector”, “T5 128-bit SIMD vector”, and “T5 64-bit SIMD vector” variants: is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#).

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in “imm4H:imm4L”. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 floating-point instructions](#).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<dt>

The data type, encoded in “cmode”:

cmode	<dt>
110x	I32
1110	I8
1111	F32

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if esize <= 32 then
        S(d) = ZeroExtend{32}(imm);
    else
        for r = 0 to regs-1 do
            D(d+r) = ZeroExtend{64}(imm);
        end;
    end;
end;
```

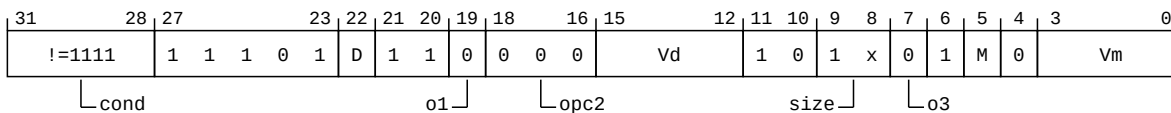
C6.1.117 VMOV (register)

Copy between FP registers copies the contents of one FP register to another.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A2](#)) and T32 ([T2](#)).

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

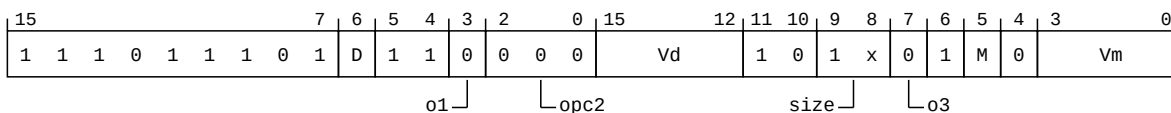
Applies when (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let single_register : boolean = (size == '10');
let advsimd : boolean = FALSE;
```

T2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let single_register : boolean = (size == '10');
let advsimd : boolean = FALSE;
```

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if single_register then
        S(d) = S(m);
    else
        D(d) = D(m);
    end;
end;
```

C6.1.118 VMOV (general-purpose register to scalar)

Copy a general-purpose register to a vector element copies a byte, halfword, or word from a general-purpose register into an Advanced SIMD scalar.

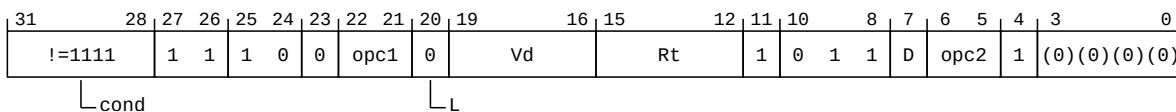
On a Floating-point-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPExc](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

```
VMOV{<c>} {<q>} {.<size>} <Dd[x]>, <Rt>
```

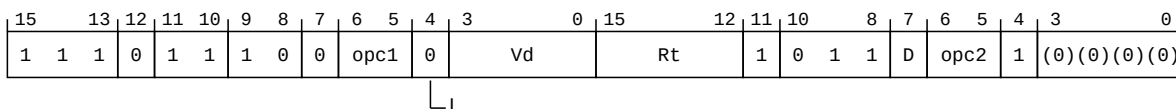
Decode for this encoding

```

let opc : bits(4) = opc1::opc2;
if opc == '0x10' then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;

```

T1



Encoding

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if opc == '0x10' then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let d : integer = UInt(D::Vd);
let t : integer = UInt(Rt);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

The data size. It must be one of:

8 Encoded as `opc1<1> = 1`. [x] is encoded in `opc1<0>`, `opc2`.

16 Encoded as `opc1<1> = 0`, `opc2<0> = 1`. [x] is encoded in `opc1<0>`, `opc2<1>`.

32 Encoded as `opc1<1> = 0`, `opc2 = 0b00`. [x] is encoded in `opc1<0>`.

omitted Equivalent to 32.

<Dd|x>

The scalar. The register <Dd> is encoded in D:Vd. For details of how [x] is encoded, see the description of <size>.

<Rt>

The source general-purpose register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    D(d)[index*esize] = R(t)[esize-1:0];
end;
```

C6.1.119 VMOV (between general-purpose register and single-precision)

Copy a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in a 32-bit SIMD&FP register to a general-purpose register, or the value held in a general-purpose register to a 32-bit SIMD&FP register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1

31	28	27	21				20	19	16	15	12		11	8	7	6	0						
!=1111				1	1	1	0	0	0	0	op	Vn			Rt		1	0	1	0	N	(0)(0)1(0)(0)(0)(0)	

└── cond

Encoding for the From general-purpose register variant

Applies when (op == 0)

VMOV{<c>}{<q>} <Sn>, <Rt>

Encoding for the To general-purpose register variant

Applies when (op == 1)

VMOV{<c>}{<q>} <Rt>, <Sn>

Decode for all variants of this encoding

```
let to_arm_register : boolean = (op == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(Vn::N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1

15										5	4	3	0	15		12		11		8	7	6				0
1	1	1	0	1	1	1	0	0	0	0	op	Vn		Rt		1		0	1	0	N	(0)(0)1(0)(0)(0)(0)				

Encoding for the From general-purpose register variant

Applies when (op == 0)

VMOV{<c>}{<q>} <Sn>, <Rt>

Encoding for the To general-purpose register variant

Applies when (op == 1)

VMOV{<c>}{<q>} <Rt>, <Sn>

Decode for all variants of this encoding

```
let to_arm_register : boolean = (op == '1');  
let t : integer = UInt(Rt);  
let n : integer = UInt(Vn::N);  
// Armv8-A removes UNPREDICTABLE for R13  
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sn>

Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Vn:N” field.

<Rt>

Is the general-purpose register that <Sn> will be transferred to or from, encoded in the “Rt” field.

Operation

```
if ConditionPassed() then  
  EncodingSpecificOperations();  
  CheckVFPEEnabled(TRUE);  
  if to_arm_register then  
    R(t) = S(n);  
  else  
    S(n) = R(t);  
  end;  
end;
```

C6.1.120 VMOV (scalar to general-purpose register)

Copy a vector element to a general-purpose register with sign or zero extension

Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

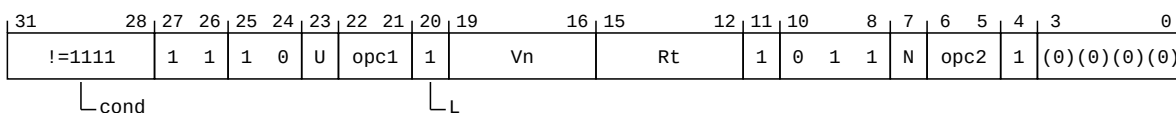
On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPXEC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



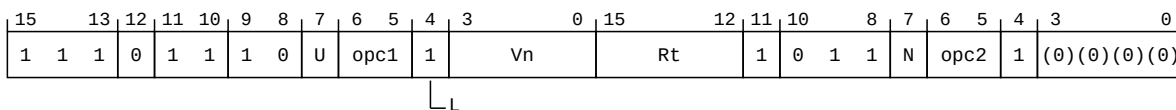
Encoding

```
VMOV{ <c> } { <q> } { .<dt> } <Rt>, <Dn[x]>
```

Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if (U::opc) IN {'10x00', 'x0x10'} then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let unsigned : boolean = (U == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(N::Vn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

T1



Encoding

```
VMOV{ <c> } { <q> } { . <dt> } <Rt>, <Dn[x]>
```


Decode for this encoding

```
let opc : bits(4) = opc1::opc2;
if (U::opc) IN {'10x00', 'x0x10'} then Undefined(); end;
let lsb : integer{} = LowestSetBit(opc[0,3]);
let esize : integer{} = 8 << lsb;
let index : integer = UInt(opc[2:lsb]);
let advsimd : boolean = (esize < 32);
let unsigned : boolean = (U == '1');
let t : integer = UInt(Rt);
let n : integer = UInt(N::Vn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

The data type. It must be one of:

S8 Encoded as U = 0, opc1<1> = 1. [x] is encoded in opc1<0>, opc2.

S16 Encoded as U = 0, opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.

U8 Encoded as U = 1, opc1<1> = 1. [x] is encoded in opc1<0>, opc2.

U16 Encoded as U = 1, opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.

32 Encoded as U = 0, opc1<1> = 0, opc2 = 0b00. [x] is encoded in opc1<0>.

omitted Equivalent to 32.

<Rt>

The destination general-purpose register.

<Dn[x]>

The scalar. For details of how [x] is encoded see the description of <dt>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if unsigned then
        R(t) = ZeroExtend{32}(D(n)[index*esize]);
    else
        R(t) = SignExtend{32}(D(n)[index*esize]);
    end;
end;
```

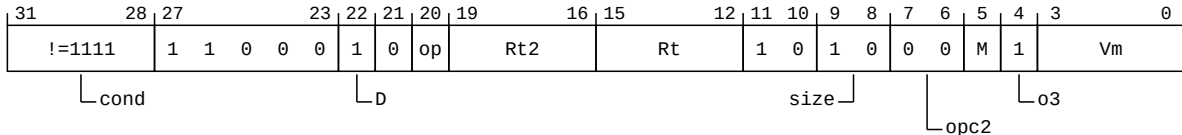
C6.1.121 VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the From general-purpose registers variant

Applies when (op == 0)

VMOV{<c>}{<q>} <Sm>, <Sml>, <Rt>, <Rt2>

Encoding for the To general-purpose registers variant

Applies when (op == 1)

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sml>

Decode for all variants of this encoding

```

let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(Vm::M);
let m2 : integer = m + 1;
let to_arm_registers : boolean = (op == '1');
if t == 15 || t2 == 15 || m == 31 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

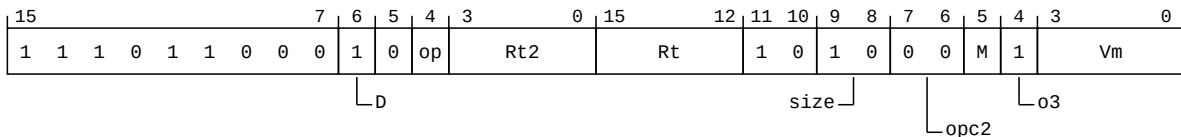
If to_arm_registers && t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If m == 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

T1



Encoding for the From general-purpose registers variant

Applies when (`op == 0`)

```
VMOV{<c>} {<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

Encoding for the To general-purpose registers variant

Applies when `(op == 1)`

```
VMOV{<c>}{<q>}    <Rt>, <Rt2>, <Sm>, <Sm1>
```

Decode for all variants of this encoding

```

let t : integer = UInt(Rt);
let t2 : integer = UInt(Rt2);
let m : integer = UInt(Vm::M);
let m2 : integer = m + 1;
let to_arm_registers : boolean = (op == '1');
if t == 15 || t2 == 15 || m == 31 then UnpredictableProcedure(); end;
if to_arm_registers && t == t2 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If to arm registers $\&\& t == t2$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The value in the destination register is **UNKNOWN**.

If $m = 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly **VMOV** (between two general-purpose registers and two single-precision registers).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sm>

Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Vm:M” field.

<Sm1>

Is the 32-bit name of the second SIMD&FP register to be transferred. This is the next SIMD&FP register after <Sm>.

<Rt>

Is the first general-purpose register that <Sm> will be transferred to or from, encoded in the “Rt” field.

<Rt2>

Is the second general-purpose register that <Sm1> will be transferred to or from, encoded in the “Rt2” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    if to_arm_registers then
        R(t) = S(m);
        R(t2) = S(m2);
    else
        S(m) = R(t);
        S(m2) = R(t2);
    end;
end;
```

C6.1.122 VMOV (register, SIMD)

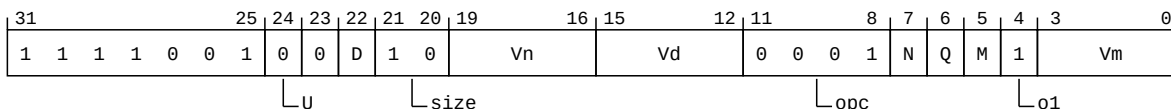
Copy between SIMD registers copies the contents of one SIMD register to another.

This is an alias of [VORR \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(register\)](#).
- The description of [VORR \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

$VMOV\{<c>\}\{<q>\}\{.<dt>\}\ \ <Dd>, \ <Dm>$

is equivalent to

$VORR\{<c>\}\{<q>\}\{.<dt>\}\ \ \{<Dd>, \ \}<Dm>, \ <Dm>$

and is the preferred disassembly when $N :: Vn == M :: Vm$.

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

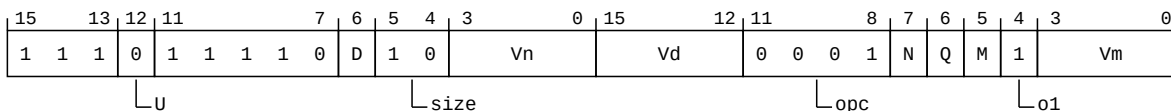
$VMOV\{<c>\}\{<q>\}\{.<dt>\}\ \ <Qd>, \ <Qm>$

is equivalent to

$VORR\{<c>\}\{<q>\}\{.<dt>\}\ \ \{<Qd>, \ \}<Qm>, \ <Qm>$

and is the preferred disassembly when $N :: Vn == M :: Vm$.

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

$VMOV\{<c>\}\{<q>\}\{.<dt>\}\ \ <Dd>, \ <Dm>$

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

and is the preferred disassembly when $N :: V_n == M :: V_m$.

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

and is the preferred disassembly when $N :: V_n == M :: V_m$.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. <dt> must not be F64, but it is otherwise ignored.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “N:Vn” and “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “N:Vn” and “M:Vm” field as <Qm>*2.

Operation

The description of [VORR \(register\)](#) gives the operational pseudocode for this instruction.

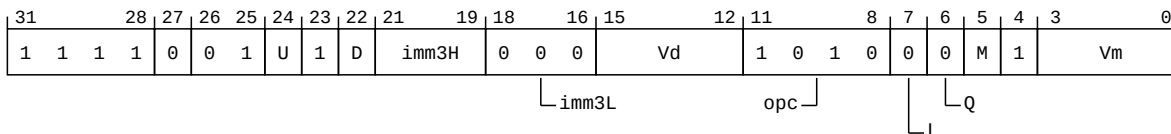
C6.1.123 VMOVL

Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

Applies when (imm3H IN {001, 010, 100})

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

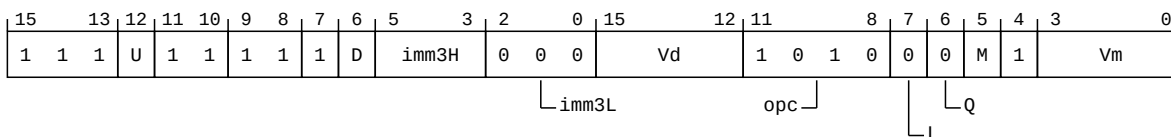
Decode for this encoding

```

if imm3H == '000' then See("Related encodings"); end;
if ! imm3H IN {'001', '010', '100'} then See("VSHLL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm3H);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let unsigned : boolean = (U == '1');

```

T1



Encoding

Applies when (imm3H IN {001, 010, 100})

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

Decode for this encoding

```
if imm3H == '000' then See("Related encodings"); end;
if ! imm3H IN {'001', '010', '100'} then See("VSHLL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm3H);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let unsigned : boolean = (U == '1');
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “U:imm3H”:

U	imm3H	<dt>
0	001	S8
0	010	S16
0	100	S32
1	001	U8
1	010	U16
1	100	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for e = 0 to elements-1 do
    let op1elt : bits(esize) = Din(m)[e*:esize];
    let result : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
  end;
end;
```


C6.1.124 VMOVN

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instructions [VRSHRN \(zero\)](#) and [VSHRN \(zero\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	7	6	5	4	3	0
1	1	1	1	0	0	1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	0	M	0	Vm
opc1														opc2				Q					

Encoding

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for this encoding

```

if size == '11' then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	7	6	5	4	3	0	
1	1	1	1	1	1	1	1	D	1	1	size	1	0		Vd		0	0	1	0	0	0	M	0	Vm
opc1														opc2				Q							

Encoding

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for this encoding

```

if size == '11' then Undefined(); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I 16
01	I 32
10	I 64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        D(d)[e*:esize] = Qin(m>>1)[e*:(2*esize)][esize-1:0];
    end;
end;
```

C6.1.125 VMRS

Move SIMD&FP Special register to general-purpose register moves the value of an Advanced SIMD and floating-point System register to a general-purpose register. When the specified System register is the **FPSCR**, a form of the instruction transfers the **FPSCR**.{N, Z, C, V} condition flags to the **APSR**.{N, Z, C, V} condition flags.

Depending on settings in the **CPACR**, **NSACR**, **HCPTR**, and **FPEXC** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

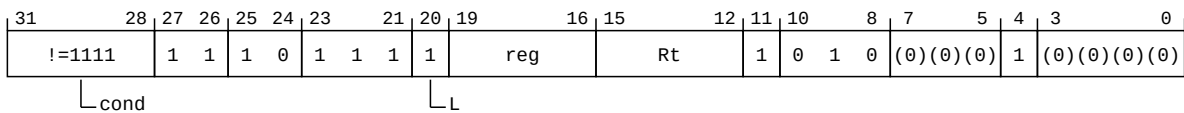
When these settings permit the execution of Advanced SIMD and floating-point instructions, if the specified floating-point System register is not the **FPSCR**, the instruction is UNDEFINED if executed in User mode.

In an implementation that includes EL2, when **HCR.TID0** is set to 1, any **VMRS** access to **FPSID** from a Non-secure EL1 mode that would be permitted if **HCR.TID0** was set to 0 generates a Hyp Trap exception. For more information, see [EL2 configurable controls](#).

For simplicity, the **VMRS** pseudocode does not show the possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



Encoding

VMRS{<c>}{<q>} <Rt>, <spec_reg>

Decode for this encoding

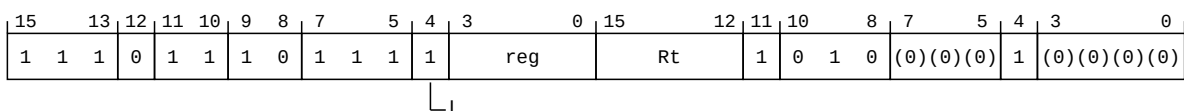
```
let t : integer = UInt(Rt);
if ! reg IN {'000x', '0101', '011x', '1000'} then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 && reg != '0001' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If ! reg IN {'000x', '0101', '011x', '1000'}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the **APSR**.{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the **Rt** field, R0 - R14.

T1



Encoding

VMRS{<c>}{<q>} <Rt>, <spec_reg>

Decode for this encoding

```
let t : integer = UInt(Rt);
if ! reg IN {'000x', '0101', '011x', '1000'} then UnpredictableProcedure(); end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 && reg != '0001' then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If ! reg IN {'000x', '0101', '011x', '1000'}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the APSR.{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Rt>

Is the general-purpose destination register, encoded in the “Rt” field. Is one of:

R0-R14 General-purpose register.

APSR_nzcv Permitted only when <spec_reg> is FPSCR. Encoded as 0b1111. The instruction transfers the **FPSCR**.{N, Z, C, V} condition flags to the **APSR**.{N, Z, C, V} condition flags.

<spec_reg>

Is the source Advanced SIMD and floating-point System register, encoded in “reg”:

reg	<spec_reg>
0000	FPSID
0001	FPSCR
001x	UNPREDICTABLE
0100	UNPREDICTABLE
0101	MVFR2
0110	MVFR1
0111	MVFR0
1000	FPEXC

reg	<spec_reg>
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then // FPSCR
        CheckVFPEnabled(TRUE);
        if t == 15 then
            PSTATE.[N,Z,C,V] = FPSCR().[N,Z,C,V];
        else
            R(t) = FPSCR();
        end;
    elseif PSTATE.EL == ELO then
        Undefined(); // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE); // Non-FPSCR registers are not affected by FPEXC.EN
        AArch32_CheckAdvSIMDorFPRegisterTraps(reg);
        case reg of
            when '0000' => R(t) = FPSID();
            when '0101' => R(t) = MVFR2();
            when '0110' => R(t) = MVFR1();
            when '0111' => R(t) = MVFR0();
            when '1000' => R(t) = FPEXC();
            otherwise => unreachable; // Dealt with above or in encoding-specific pseudocode
        end;
    end;
end;
```

C6.1.126 VMSR

Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register.

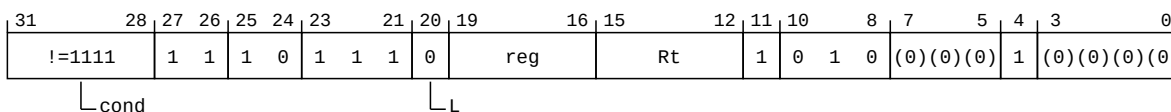
Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

When these settings permit the execution of Advanced SIMD and floating-point instructions:

- If the specified floating-point System register is [FPSID](#) or [FPEXC](#), the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the [FPSID](#) and the instruction is executed in a mode other than User mode, the instruction is ignored.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VMSR{<c>}{<q>} <spec_reg>, <Rt>

Decode for this encoding

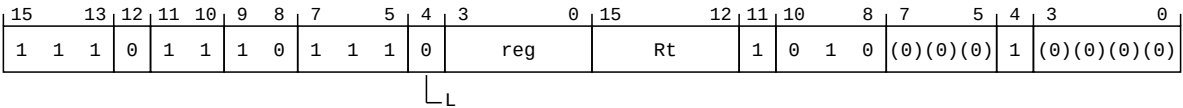
```
let t : integer = UInt(Rt);
if reg != '000x' && reg != '1000' then
    let c : Constraint = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF =>
            Undefined();
        when Constraint_NOP =>
            ExecuteAsNOP();
    end;
end;
end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

T1



Encoding

VMSR{<c>}{<q>} <spec_reg>, <Rt>

Decode for this encoding

```
let t : integer = UInt(Rt);
if reg != '000x' && reg != '1000' then
  let c : Constraint = ConstrainUnpredictable(Unpredictable_VMSR);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF =>
      Undefined();
    when Constraint_NOP =>
      ExecuteAsNOP();
  end;
end;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If reg != '000x' && reg != '1000', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<spec_reg>

Is the destination Advanced SIMD and floating-point System register, encoded in “reg”:

reg	<spec_reg>
0000	FPSID

reg	<spec_reg>
0001	FPSCR
001x	UNPREDICTABLE
01xx	UNPREDICTABLE
1000	FPEXC
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

<Rt>

Is the general-purpose source register, encoded in the “Rt” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                                // FPSCR
        CheckVFPEEnabled(TRUE);
        FPSCR() = R(t);
    elsif PSTATE.EL == ELO then
        Undefined();                                     // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEEnabled(FALSE);                         // Non-FPSCR registers are not affected by FPEXC.EN
        case reg of
            when '0000' =>                                // VMSR access to FPSID is ignored
                pass;
            when '1000' =>
                FPEXC() = R(t);
            otherwise =>
                unreachable; // Dealt with above or in encoding-specific pseudocode
        end;
    end;
end;
```


C6.1.127 VMUL (floating-point)

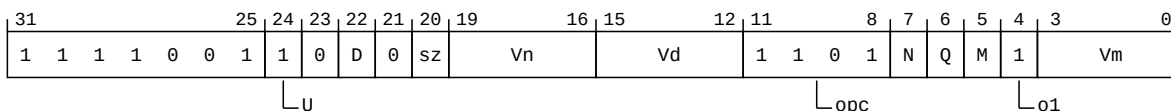
Vector Multiply (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

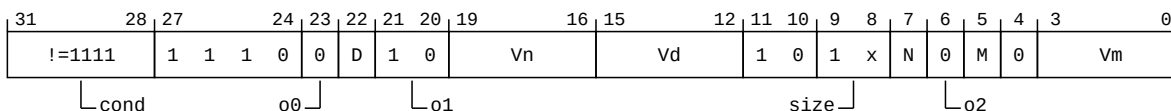
Applies when ($Q == 1$)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

A2



Encoding for the Single-precision scalar variant

Applies when ($size == 10$)

```
VMUL{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VMUL{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

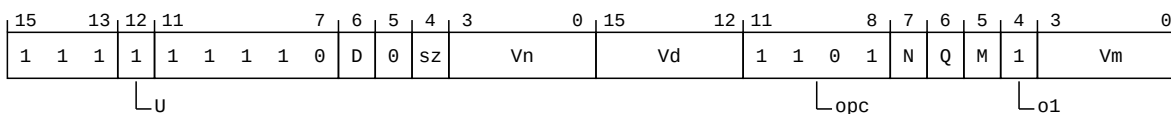
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

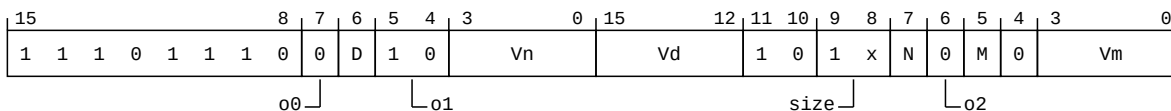
```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VMUL{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VMUL{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*:esize] = FPMul{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
            when 32 =>
                S(d) = FPMul{32}(S(n), S(m), fpcr);
            when 64 =>
                D(d) = FPMul{64}(D(n), D(m), fpcr);
        end;
    end;
end;

```

C6.1.128 VMUL (integer and polynomial)

Vector Multiply (integer and polynomial)

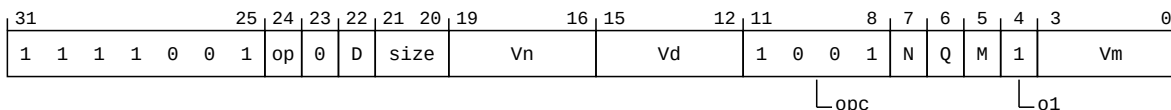
Vector Multiply multiplies corresponding elements in two vectors.

For information about multiplying polynomials, see Polynomial arithmetic over {0, 1}.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

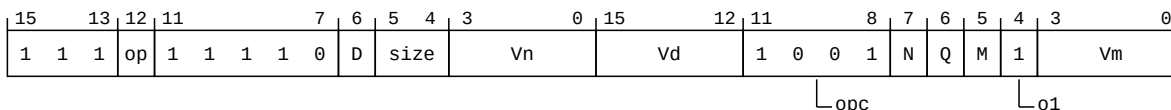
Decode for all variants of this encoding

```

if size == '11' || (op == '1' && size != '00') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let polynomial : boolean = (op == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if size == '11' || (op == '1' && size != '00') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let polynomial : boolean = (op == '1');
let long_destination : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “op:size”:

op	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	00	P8

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            var product : bits(2 * esize);
            if polynomial then
                product = PolynomialMult{esize, esize}(op1elt, op2elt);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                product = (element1 * element2)[2*esize-1:0];
            end;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = product;
            else
                D(d+r)[e*:esize] = product[esize-1:0];
            end;
        end;
    end;
end;
```

C6.1.129 VMUL (by scalar)

Vector Multiply (by scalar)

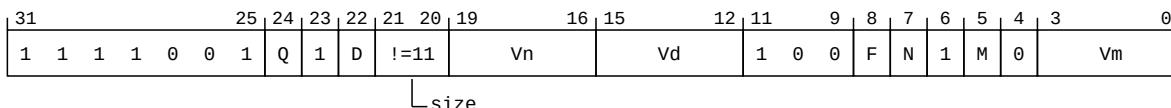
Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, <Dn>, <Dm>[<index>]}

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, <Qn>, <Dm>[<index>]}

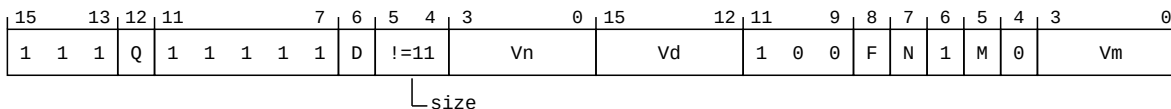
Decode for all variants of this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>[<index>]

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || (F == '1' && size == '01') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = FALSE; // "Don't care" value: TRUE produces same functionality
let floating_point : boolean = (F == '1');
let long_destination : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If F == '1' && size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register. When <dt> is I16 or F16, this is encoded in the “Vm<2:0>” field. Otherwise it is encoded in the “Vm” field.

<index>

Is the element index. When <dt> is I16 or F16, this is in the range 0 to 3 and is encoded in the “M:Vm<3>” field. Otherwise it is in the range 0 to 1 and is encoded in the “M” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*esize];
            if floating_point then
                D(d+r)[e*esize] = FPMul{esize}(op1elt, op2elt, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                if long_destination then
                    Q(d>>1)[e*(2*esize)] = (element1 * element2)[2*esize-1:0];
                else
                    D(d+r)[e*esize] = (element1 * element2)[esize-1:0];
                end;
            end;
        end;
    end;
end;
end;
end;

```

C6.1.130 VMULL (integer and polynomial)

Vector Multiply Long (integer and polynomial)

Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see Polynomial arithmetic over {0, 1}.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0	
1	1	1	1	0	0	1	U	1	D	!=11	Vn		Vd		1	1	op	0	N	0	M	0	Vm	

└─ size

Encoding

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if op == '1' && (U == '1' || size == '01') then Undefined(); end;
if op == '1' && size == '10' && !IsFeatureImplemented(FEAT_PMULL) then Undefined(); end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let polynomial : boolean = (op == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = if polynomial && size == '10' then 64 else 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;

```

T1

15	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	U	1	1	1	1	1	D	!=11	Vn		Vd		1	1	op	0	N	0	M	0	Vm	

└─ size

Encoding

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if op == '1' && (U == '1' || size == '01') then Undefined(); end;
if op == '1' && size == '10' && InITBlock() then UnpredictableProcedure(); end;
if op == '1' && size == '10' && !IsFeatureImplemented(FEAT_PMULL) then
    UnpredictableProcedure();
end;
if Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let polynomial : boolean = (op == '1');
let long_destination : boolean = TRUE;
let esize : integer{} = if polynomial && size == '10' then 64 else 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;
let regs : integer = 1;
```

CONSTRAINED UNPREDICTABLE behavior

If `op == '1' && size == '10' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “op:U:size”:

op	U	size	<dt>
0	0	00	S8
0	0	01	S16
0	0	10	S32
0	1	00	U8
0	1	01	U16
0	1	10	U32
1	0	00	P8

op	U	size	<dt>
1	0	10	P64

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            let op2elt : bits(esize) = Din(m+r)[e*:esize];
            var product : bits(2 * esize);
            if polynomial then
                product = PolynomialMult{esize, esize}(op1elt, op2elt);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                product = (element1 * element2)[2*esize-1:0];
            end;
            if long_destination then
                Q(d>>1)[e*:(2*esize)] = product;
            else
                D(d+r)[e*:esize] = product[esize-1:0];
            end;
        end;
    end;
end;

```

C6.1.131 VMULL (by scalar)

Vector Multiply Long (by scalar)

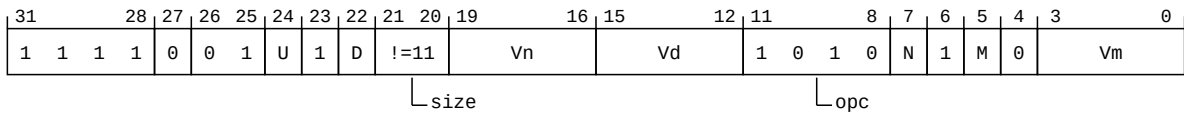
Vector Multiply Long multiplies each element in a vector by a scalar, and places the results in a second vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

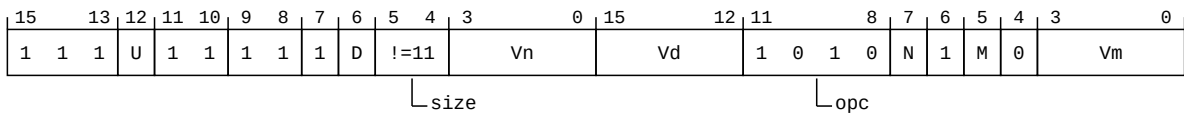
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let floating_point : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);

```

T1



Encoding

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let unsigned : boolean = (U == '1');
let long_destination : boolean = TRUE;
let floating_point : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let regs : integer = 1;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>” field when <dt> is S16 or U16, otherwise the “Vm” field.

<index>

Is the element index in the range 0 to 3, encoded in the “M:Vm<3>” field when <dt> is S16 or U16, otherwise in range 0 to 1, encoded in the “M” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    let op2elt : bits(esize) = Din(m)[index*:esize];
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = Din(n+r)[e*:esize];
            if floating_point then
                D(d+r)[e*:esize] = FPMul{esize}(op1elt, op2elt, fpcr);
            else
                let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
                let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
                if long_destination then
                    Q(d>>1)[e*:(2*esize)] = (element1 * element2)[2*esize-1:0];
                else
                    D(d+r)[e*:esize] = (element1 * element2)[esize-1:0];
                end;
            end;
        end;
    end;
end;
end;
end;

```

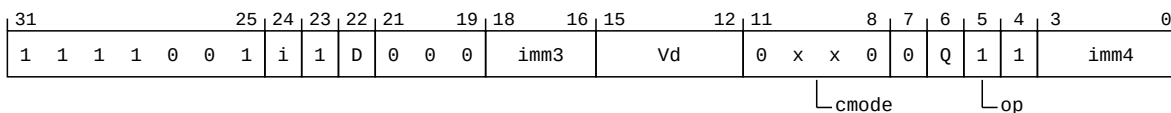
C6.1.132 VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#), [A2](#), and [A3](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

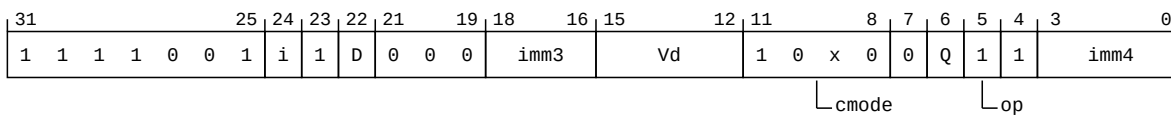
Applies when (Q == 1)

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VMVN{<c>}{<q>}.I16 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VMVN{<c>}{<q>}.I16 <Qd>, #<imm>
```

Decode for all variants of this encoding

```

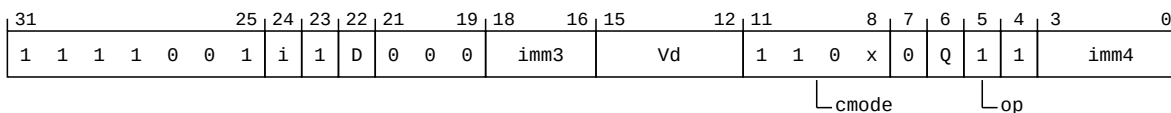
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
  See("Related encodings");
end;

if Q == '1' && Vd[0] == '1' then Undefined(); end;

let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;

```

A3



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```

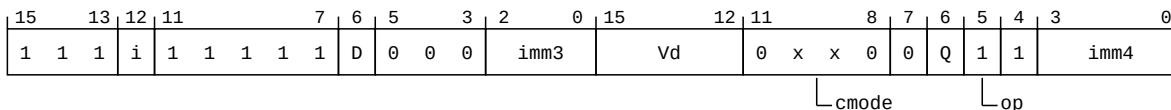
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
  See("Related encodings");
end;

if Q == '1' && Vd[0] == '1' then Undefined(); end;

let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

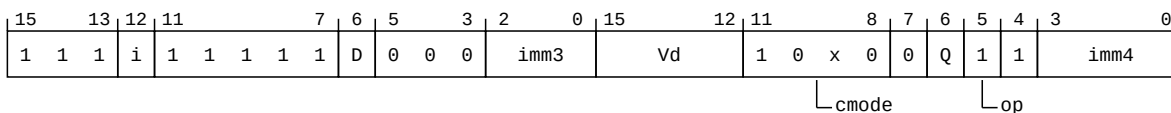
Applies when (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

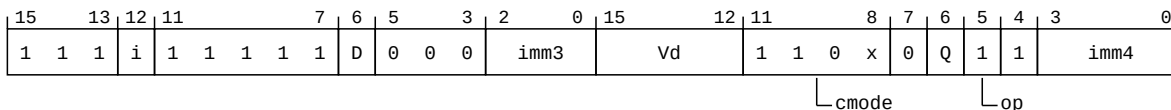
Applies when (Q == 1)

VMVN{<c>}{<q>}.I16 <Qd>, #<imm>

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T3



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

Decode for all variants of this encoding

```
if (cmode[0] == '1' && cmode[3:2] != '11') || cmode[3:1] == '111' then
    See("Related encodings");
end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('1', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, “A2 64-bit SIMD vector”, “A3 128-bit SIMD vector”, and “A3 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, “T2 64-bit SIMD vector”, “T3 128-bit SIMD vector”, and “T3 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = NOT(imm64);
    end;
end;
```

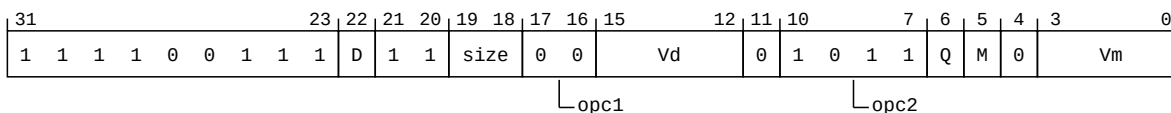
C6.1.133 VMVN (register)

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>

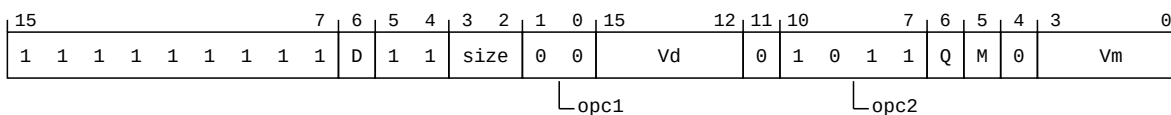
Decode for all variants of this encoding

```

if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>

Decode for all variants of this encoding

```

if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = NOT(D(m+r));
    end;
end;

```

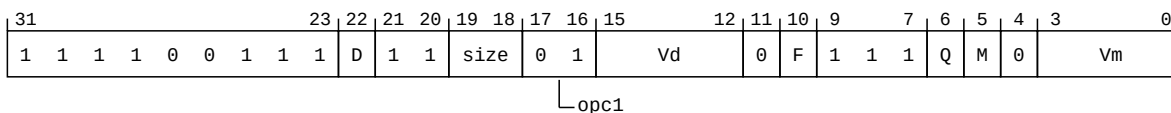
C6.1.134 VNEG

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

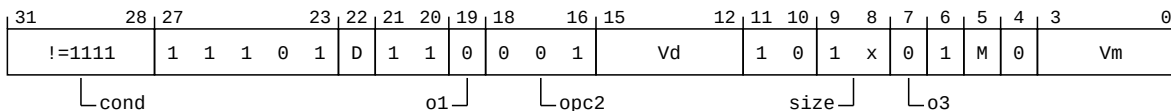
Decode for all variants of this encoding

```

if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNEG{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNEG{<c>} {<q>} .F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '1l' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '1l' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

T1

15								7	6	5	4	3	2	1	0	15				12	11	10	9	7			6	5	4	3	0			
1	1	1	1	1	1	1	1	1	D	1	1	size	0	1	Vd				0	F	1	1	1	Q	M	0	Vm							
															└─opc1																			

Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VNEG}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \langle Qd \rangle, \quad \langle Qm \rangle$$

Decode for all variants of this encoding

```

if size == '11' || (F == '1' && size != '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let advsimd : boolean = TRUE;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

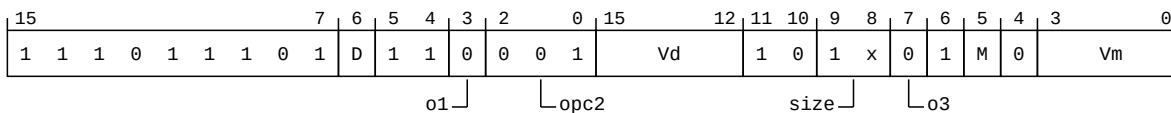
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

VNEG{<c>}{<q>}.F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

VNEG{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let floating_point : boolean = ARBITRARY : boolean;
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                if floating_point then
                    D(d+r)[e*:esize] = FPNeg(esize)(D(m+r)[e*:esize], fpcr);
                else
                    let result : integer = -SInt(D(m+r)[e*:esize]);
                    D(d+r)[e*:esize] = result[esize-1:0];
                end;
            end;
        end;
    else // VFP instruction
        let fpcr : FPCR_Type = EffectiveFPCR();
        case esize of
```

```
        when 32 => S(d) = FPNeg{32}(S(m), fpcr);  
        when 64 => D(d) = FPNeg{64}(D(m), fpcr);  
    end;  
end;  
end;
```

C6.1.135 VNMLA

Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

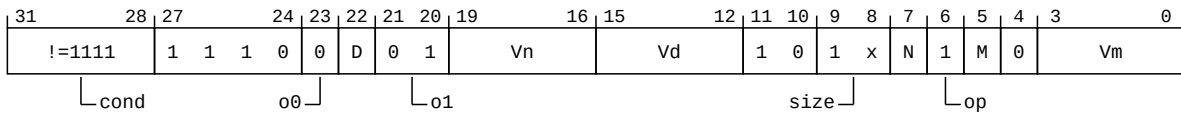
Note

Arm recommends that software does not use the VNMLA instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

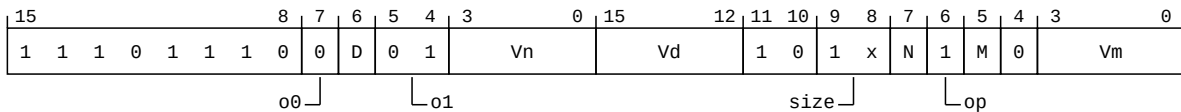
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer() = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D:D) else UInt(Vd:D);
let n : integer = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
let m : integer = if size == '11' then UInt(M:Vm) else UInt(Vm:M);

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

Assembler Symbols

 $\langle c \rangle$

See [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 =>
            let product32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr),
                                                            FPNeg{32}(product32, fpcr), fpcr);
                when VFPNegMul_VNMLS => S(d) = FPAdd{32}(FPNeg{32}(S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL => S(d) = FPNeg{32}(product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr),
                                                            FPNeg{64}(product64, fpcr), fpcr);
                when VFPNegMul_VNMLS => D(d) = FPAdd{64}(FPNeg{64}(D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL => D(d) = FPNeg{64}(product64, fpcr);
            end;
    end;
end;
```

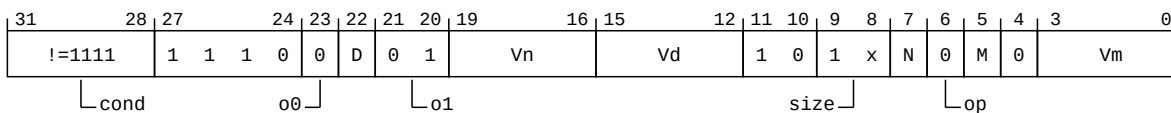
C6.1.136 VNMLS

Vector Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

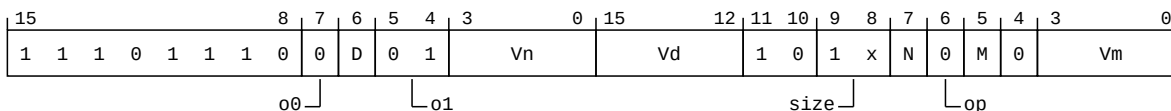
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let vtype : VFPNegMul = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 =>
            let product32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => S(d) = FPAAdd{32}(FPNeg{32}(S(d), fpcr),
                                                            FPNeg{32}(product32, fpcr), fpcr);
                when VFPNegMul_VNMLS => S(d) = FPAAdd{32}(FPNeg{32}(S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL => S(d) = FPNeg{32}(product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => D(d) = FPAAdd{64}(FPNeg{64}(D(d), fpcr),
                                                            FPNeg{64}(product64, fpcr), fpcr);
                when VFPNegMul_VNMLS => D(d) = FPAAdd{64}(FPNeg{64}(D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL => D(d) = FPNeg{64}(product64, fpcr);
            end;
    end;
end;

```

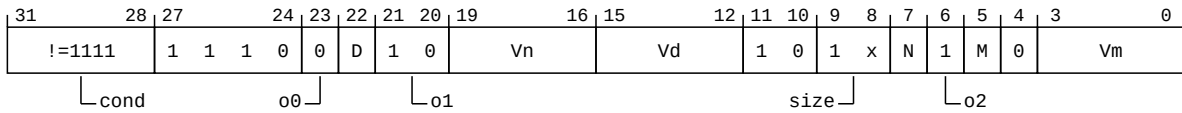
C6.1.137 VNMUL

Vector Negate Multiply multiplies together two floating-point register values, and writes the negation of the result to the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMUL{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMUL{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

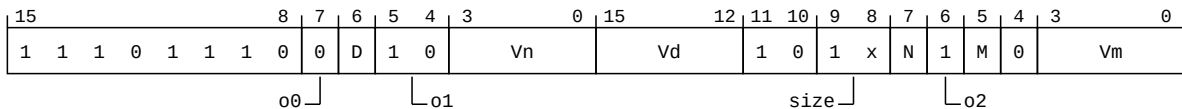
```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let vtype : VFPNegMul = VFPNegMul_VNMUL;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VNMUL{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VNMUL{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let vtype : VFPNegMul = VFPNegMul_VNMUL;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 =>
            let product32 : bits(32) = FPMul{}(S(n), S(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => S(d) = FPAAdd{32}(FPNeg{32}(S(d), fpcr),
                                                            FPNeg{32}(product32, fpcr), fpcr);
                when VFPNegMul_VNMLS => S(d) = FPAAdd{32}(FPNeg{32}(S(d), fpcr), product32, fpcr);
                when VFPNegMul_VNMUL => S(d) = FPNeg{32}(product32, fpcr);
            end;
        when 64 =>
            let product64 : bits(64) = FPMul{}(D(n), D(m), fpcr);
            case vtype of
                when VFPNegMul_VNMLA => D(d) = FPAAdd{64}(FPNeg{64}(D(d), fpcr),
                                                            FPNeg{64}(product64, fpcr), fpcr);
                when VFPNegMul_VNMLS => D(d) = FPAAdd{64}(FPNeg{64}(D(d), fpcr), product64, fpcr);
                when VFPNegMul_VNMUL => D(d) = FPNeg{64}(product64, fpcr);
            end;
    end;
end;

```

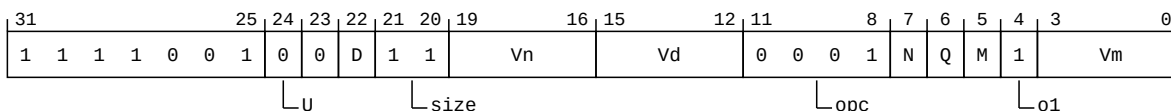
C6.1.138 VORN (register)

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

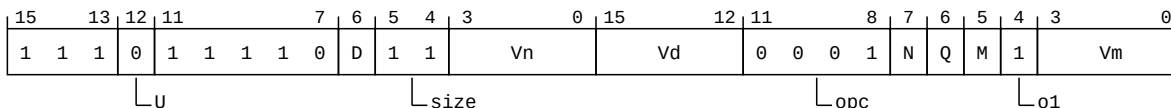
Applies when (Q == 1)

VORN{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORN{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) OR NOT(D(m+r));
    end;
end;

```

C6.1.139 VORN (immediate)

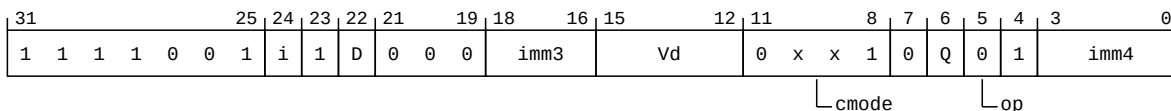
Vector Bitwise OR NOT (immediate) performs a bitwise OR between a register value and the complement of an immediate value, and returns the result into the destination vector.

This is a pseudo-instruction of [VORR \(immediate\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(immediate\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VORR \(immediate\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I32 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

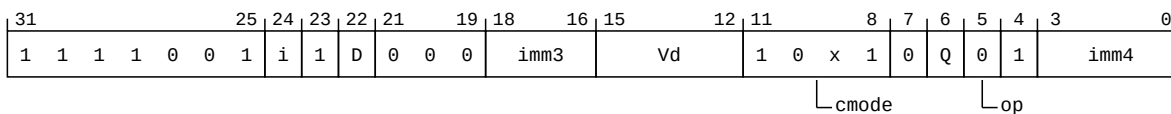
Applies when (Q == 1)

VORN{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I32 <Qd>, #~<imm>

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I16 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

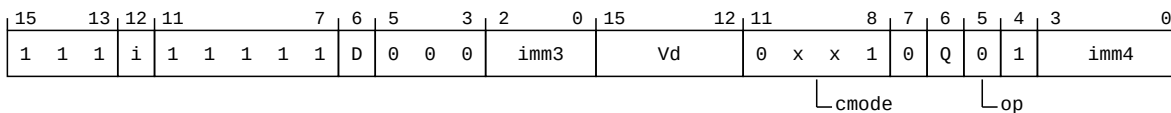
Applies when (Q == 1)

VORN{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I16 <Qd>, #~<imm>

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I32 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

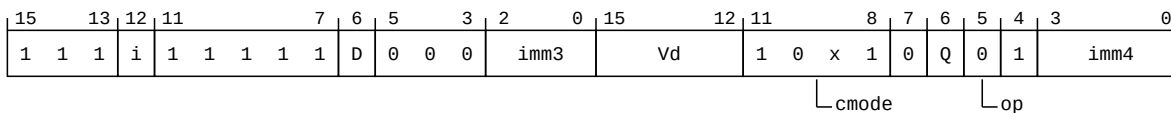
Applies when (Q == 1)

VORN{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I32 <Qd>, #~<imm>

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I16 <Dd>, #~<imm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VORN{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I16 <Qd>, #~<imm>

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

Operation

The description of [VORR \(immediate\)](#) gives the operational pseudocode for this instruction.

C6.1.140 VORR (immediate)

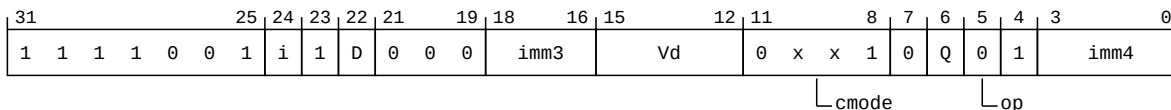
Vector Bitwise OR (immediate) performs a bitwise OR between a register value and an immediate value, and returns the result into the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VORN \(immediate\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VORR{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

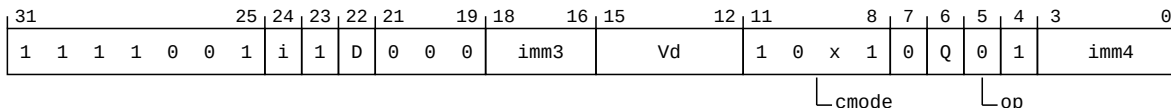
Applies when (Q == 1)

```
VORR{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VORR{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

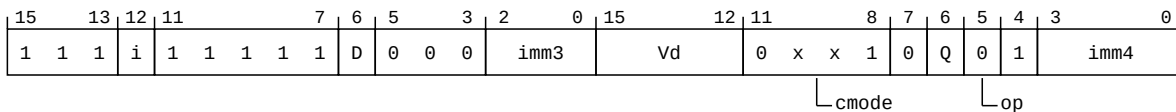
Applies when (Q == 1)

```
VORR{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VORR{<c>}{<q>}.I32 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

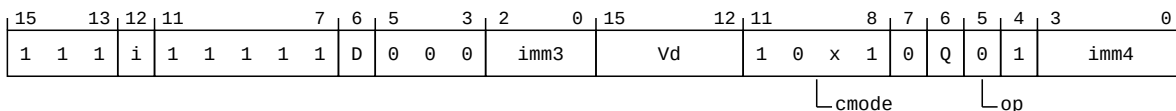
Applies when (Q == 1)

```
VORR{<c>}{<q>}.I32 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```
if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;
```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VORR{<c>}{<q>}.I16 {<Dd>, }<Dd>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VORR{<c>}{<q>}.I16 {<Qd>, }<Qd>, #<imm>
```

Decode for all variants of this encoding

```

if cmode[0] == '0' || cmode[3:2] == '11' then See("VMOV (immediate)"); end;
if Q == '1' && Vd[0] == '1' then Undefined(); end;
let imm64 : bits(64) = AdvSIMDEExpandImm('0', cmode, i::imm3::imm4);
let d : integer = UInt(D::Vd);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<imm>

Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#).

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(d+r) OR imm64;
    end;
end;

```

C6.1.141 VORR (register)

Vector bitwise OR (register) performs a bitwise OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

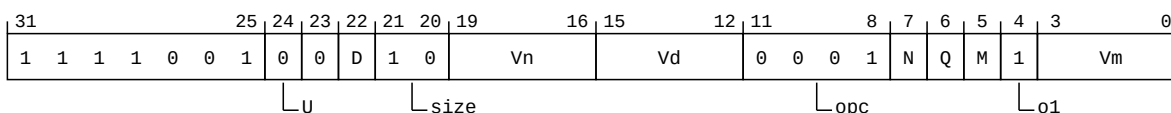
Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the alias [VMOV \(register, SIMD\)](#).

This instruction is used by the pseudo-instructions [VRSHR \(zero\)](#) and [VSHR \(zero\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

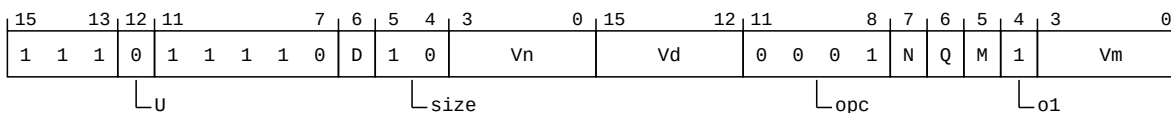
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>
For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
- <q>
See [Standard assembler syntax fields](#).
- <dt>
An optional data type. It is ignored by assemblers, and does not affect the encoding.
- <Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.
- <Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.
- <Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.
- <Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.
- <Qn>
Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.
- <Qm>
Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Alias Conditions

Alias	Is preferred when
VMOV (register, SIMD)	N : Vn == M : Vm
VRSHR (zero)	Never

Alias	Is preferred when
VSHR (zero)	Never

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        D(d+r) = D(n+r) OR D(m+r);
    end;
end;
```


C6.1.142 VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

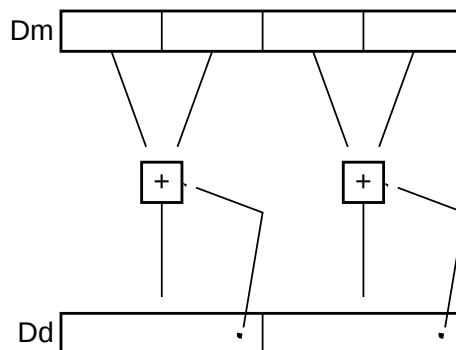
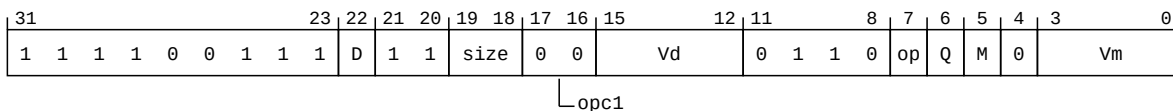


Figure C6-2 VPADAL doubleword operation for data type S16

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

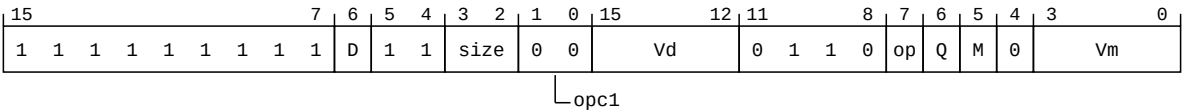
Applies when (Q == 1)

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “op:size”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
x	11	RESERVED
1	00	U8
1	01	U16

op	size	<dt>
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        for e = 0 to h-1 do
            let op1elt : bits(esize) = D(m+r)[(2*e)*:esize];
            let op2elt : bits(esize) = D(m+r)[(2*e+1)*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = element1 + element2;
            D(d+r)[e*:(2*esize)] = D(d+r)[e*:(2*esize)] + result;
        end;
    end;
end;
```

C6.1.143 VPADD (floating-point)

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

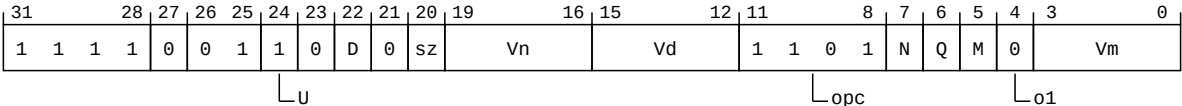
The operands and result are doubleword vectors.

The operand and result elements are floating-point numbers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

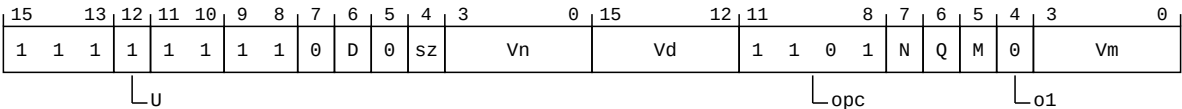
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if Q == '1' then Undefined(); end;
if sz == '1' then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
    
```

T1



Encoding

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if Q == '1' then Undefined(); end;
if sz == '1' then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
    
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

Assembler Symbols

`<c>`

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

`<q>`

See [Standard assembler syntax fields](#).

`<dt>`

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

`<Dd>`

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

`<Dn>`

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

`<Dm>`

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        dest[e*:esize] = FPAdd(esize) (D(n) [(2*e)*:esize], D(n) [(2*e+1)*:esize], fpcr);
        dest[(e+h)*:esize] = FPAdd(esize) (D(m) [(2*e)*:esize], D(m) [(2*e+1)*:esize], fpcr);
    end;

    D(d) = dest;
end;
```

C6.1.144 VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

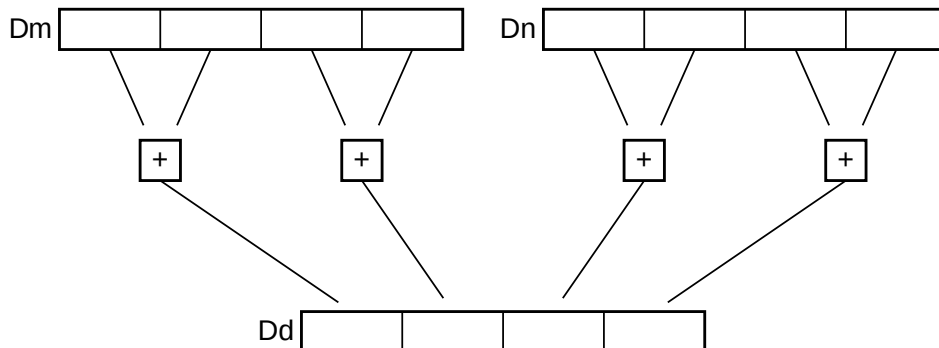
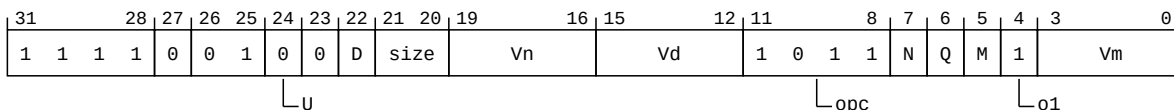


Figure C6-3 VPADD doubleword operation for data type I16

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

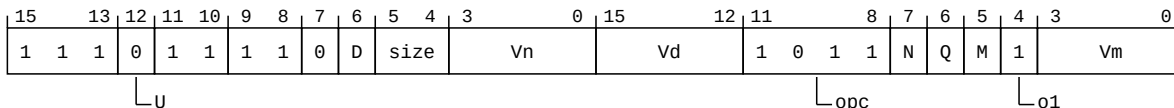
Decode for this encoding

```

if size == 'l1' || Q == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1



Encoding

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if size == '11' || Q == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>
For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>
See [Standard assembler syntax fields](#).

<dt>
Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  var dest : bits(64);
  let h : integer = elements DIV 2;

  for e = 0 to h-1 do
    dest[e*:esize] = D(n)[(2*e)*:esize] + D(n)[(2*e+1)*:esize];
    dest[(e+h)*:esize] = D(m)[(2*e)*:esize] + D(m)[(2*e+1)*:esize];
  end;

  D(d) = dest;
end;
```

C6.1.145 VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

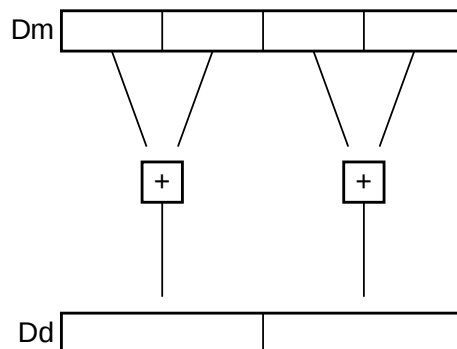
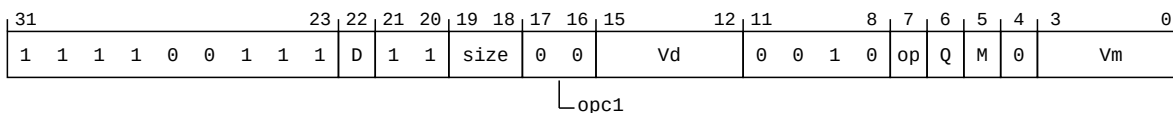


Figure C6-4 VPADDL doubleword operation for data type S16

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>

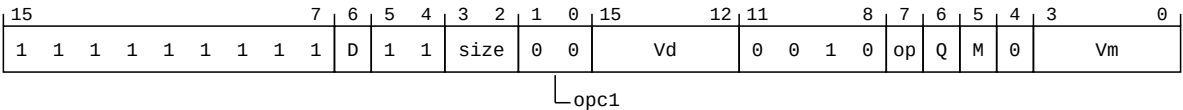
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```


T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “op:size”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
x	11	RESERVED
1	00	U8
1	01	U16

op	size	<dt>
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        for e = 0 to h-1 do
            let op1elt : bits(esize) = D(m+r)[(2*e)*:esize];
            let op2elt : bits(esize) = D(m+r)[(2*e+1)*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = element1 + element2;
            D(d+r)[e*(2*esize):] = result[2*esize-1:0];
        end;
    end;
end;
```

C6.1.146 VPMAX (floating-point)

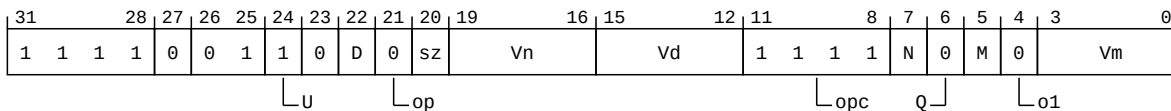
Vector Pairwise Maximum (floating-point)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

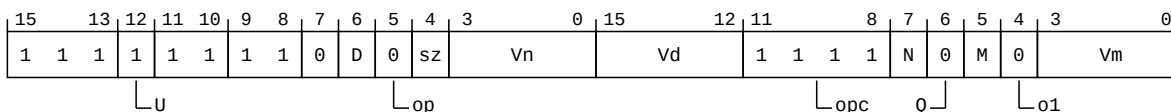
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
  
```

T1



Encoding

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
  
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

Assembler Symbols

`<c>`

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

`<q>`

See [Standard assembler syntax fields](#).

`<dt>`

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

`<Dd>`

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

`<Dn>`

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

`<Dm>`

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let fpcr : FPCR_Type = StandardFPCR();
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        dest[e*:esize] = (if maximum then FPMMax(esize)(op1elt, op2elt, fpcr)
                        else FPMMin(esize)(op1elt, op2elt, fpcr));

        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        dest[(e+h)*:esize] = (if maximum then FPMMax(esize)(op1elt, op2elt, fpcr)
                            else FPMMin(esize)(op1elt, op2elt, fpcr));

    end;

    D(d) = dest;
end;
```

C6.1.147 VPMAX (integer)

Vector Pairwise Maximum (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

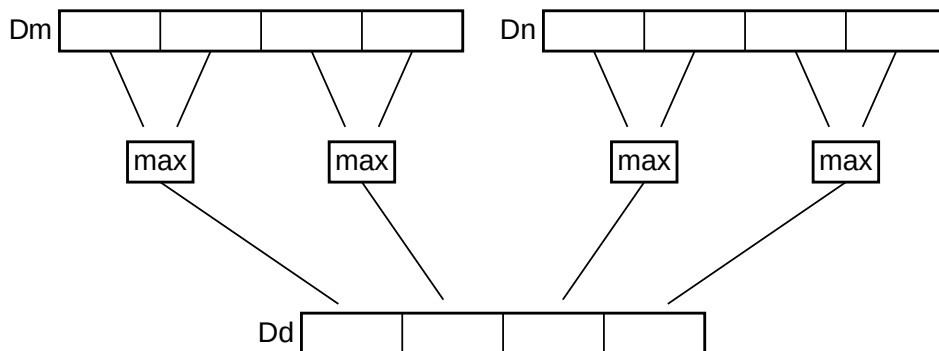


Figure C6-5 VPMAX doubleword operation for data type S16 or U16

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
1	1	1	1	0	0	1	U	0	D	size	Vn	Vd	1	0	1	0	N	0	M	0	Vm
														opc		Q		op			

Encoding

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if size == 'l1' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1

15	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	8	7	6	5	4	3	0	
1	1	1	U	1	1	1	1	0	D	size	Vn		Vd		1	0	1	0	N	0	M	0	Vm
																opc		Q		op			

Encoding

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>
For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>
See [Standard assembler syntax fields](#).

<dt>
Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>
Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  var dest : bits(64);
  let h : integer = elements DIV 2;
```

```
for e = 0 to h-1 do
    var op1elt : bits(esize) = D(n)[(2*e)*:esize];
    var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
    var element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
    var element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
    var result : integer = if maximum then Max(element1, element2) else Min(element1, element2);
    dest[e*:esize] = result[esize-1:0];
    op1elt = D(m)[(2*e)*:esize];
    op2elt = D(m)[(2*e+1)*:esize];
    element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
    element2 = if unsigned then UInt(op2elt) else SInt(op2elt);
    result = if maximum then Max(element1, element2) else Min(element1, element2);
    dest[(e+h)*:esize] = result[esize-1:0];
end;

D(d) = dest;
end;
```

C6.1.148 VPMIN (floating-point)

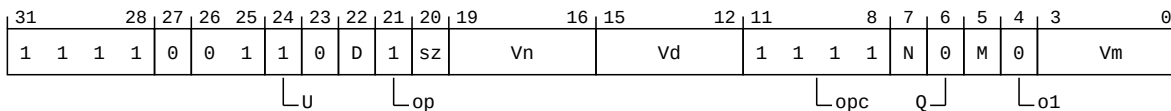
Vector Pairwise Minimum (floating-point)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

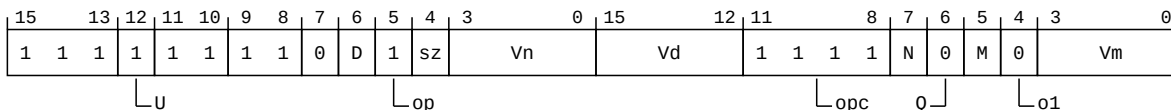
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
  
```

T1



Encoding

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if sz == '1' then Undefined(); end;
let maximum : boolean = (op == '0');
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
  
```


CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let fpcr : FPCR_Type = StandardFPCR();
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        dest[e*:esize] = (if maximum then FPCrMax(esize)(op1elt, op2elt, fpcr)
                        else FPCrMin(esize)(op1elt, op2elt, fpcr));

        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        dest[(e+h)*:esize] = (if maximum then FPCrMax(esize)(op1elt, op2elt, fpcr)
                            else FPCrMin(esize)(op1elt, op2elt, fpcr));

    end;

    D(d) = dest;
end;
```

C6.1.149 VPMIN (integer)

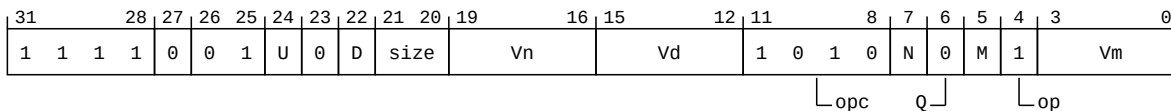
Vector Pairwise Minimum (integer)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

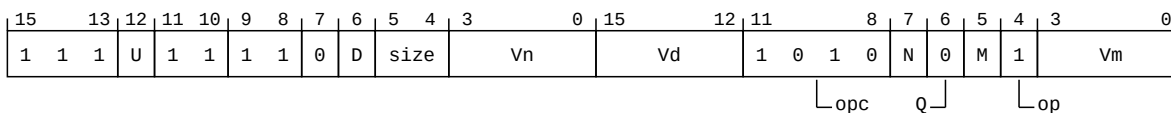
Decode for this encoding

```

if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1



Encoding

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```

if size == '11' then Undefined(); end;
let maximum : boolean = (op == '0');
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var dest : bits(64);
    let h : integer = elements DIV 2;

    for e = 0 to h-1 do
        var op1elt : bits(esize) = D(n)[(2*e)*:esize];
        var op2elt : bits(esize) = D(n)[(2*e+1)*:esize];
        var element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
        var element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        var result : integer = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[e*:esize] = result[esize-1:0];
        op1elt = D(m)[(2*e)*:esize];
        op2elt = D(m)[(2*e+1)*:esize];
        element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        element2 = if unsigned then UInt(op2elt) else SInt(op2elt);
        result = if maximum then Max(element1, element2) else Min(element1, element2);
        dest[(e+h)*:esize] = result[esize-1:0];
    end;

    D(d) = dest;
```

end;

C6.1.150 VPOP

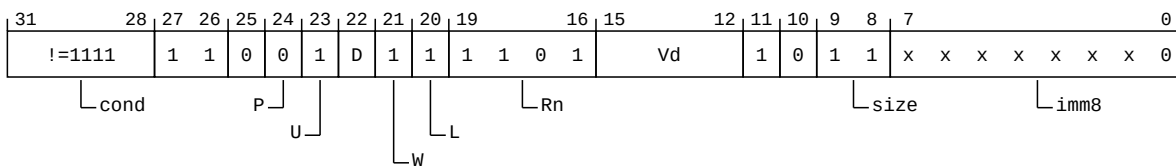
Pop SIMD&FP registers from stack loads multiple consecutive Advanced SIMD and floating-point register file registers from the stack.

This is an alias of [VLDM](#), [VLDMDB](#), [VLDMIA](#). This means:

- The encodings in this description are named to match the encodings of [VLDM](#), [VLDMDB](#), [VLDMIA](#).
- The description of [VLDM](#), [VLDMDB](#), [VLDMIA](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Increment After variant

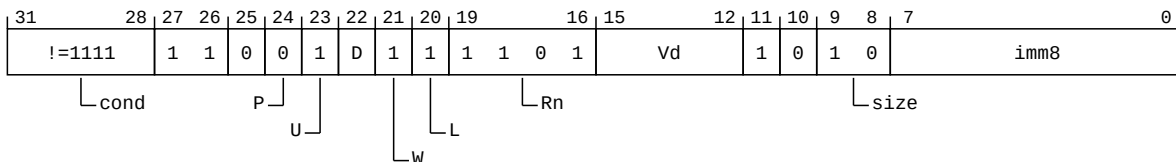
VPOP{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

A2



Encoding for the Increment After variant

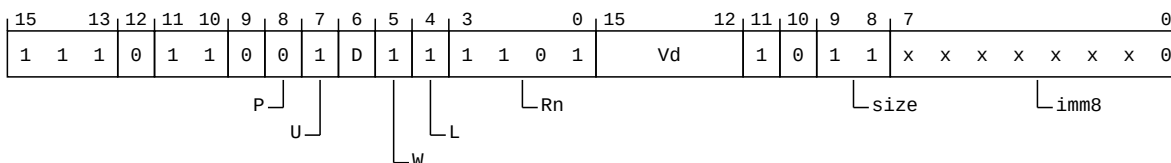
VPOP{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

T1



Encoding for the Increment After variant

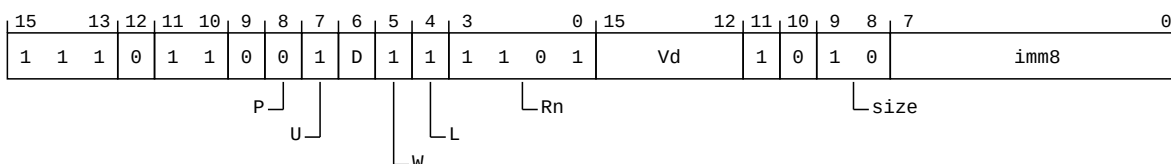
VPOP{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

T2



Encoding for the Increment After variant

VPOP{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

<sreglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “Vd:D”, and “imm8” is set to the number of registers in the list. The list must contain at least one register.

Operation

The description of [VLDM](#), [VLDMDB](#), [VLDMIA](#) gives the operational pseudocode for this instruction.

C6.1.151 VPUSH

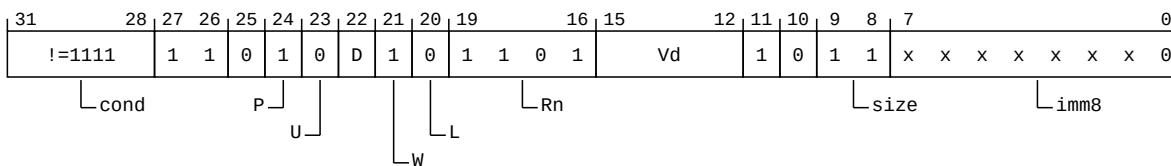
Push SIMD&FP registers to stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack.

This is an alias of [VSTM](#), [VSTMDB](#), [VSTMIA](#). This means:

- The encodings in this description are named to match the encodings of [VSTM](#), [VSTMDB](#), [VSTMIA](#).
- The description of [VSTM](#), [VSTMDB](#), [VSTMIA](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Decrement Before variant

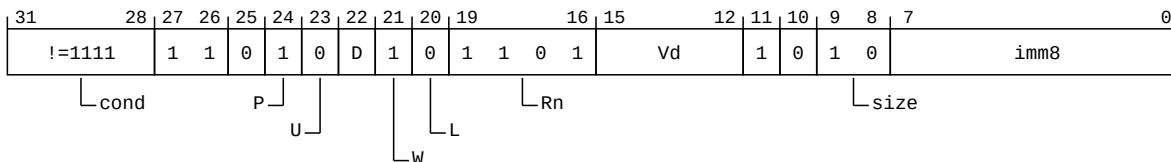
VPUSH{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

A2



Encoding for the Decrement Before variant

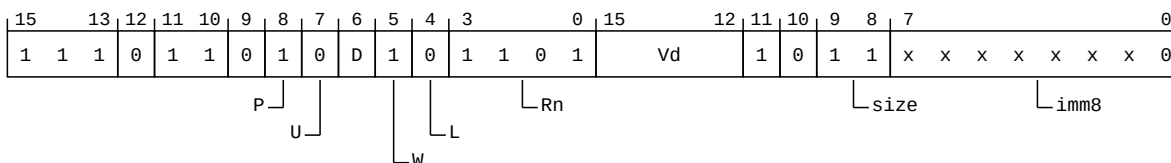
VPUSH{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

T1



Encoding for the Decrement Before variant

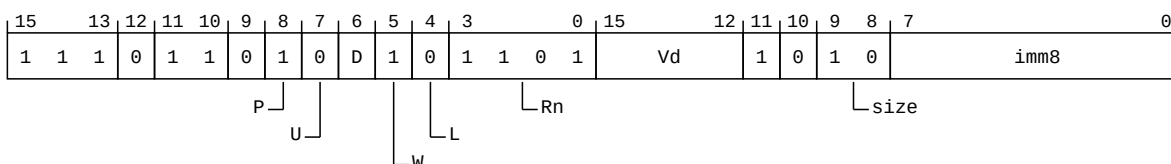
VPUSH{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

T2



Encoding for the Decrement Before variant

VPUSH{<c>}{<q>}{.<size>} <sreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>

and is always the preferred disassembly.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

<sreglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “Vd:D”, and “imm8” is set to the number of registers in the list. The list must contain at least one register.

Operation

The description of [VSTM](#), [VSTMDB](#), [VSTMIA](#) gives the operational pseudocode for this instruction.

C6.1.152 VQABS

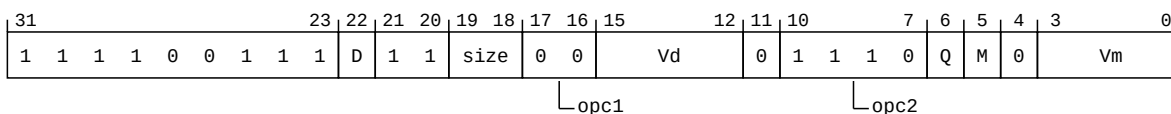
Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, **FPSCR.QC**, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VOABS{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VOABS}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \langle Qd \rangle, \quad \langle Qm \rangle$$

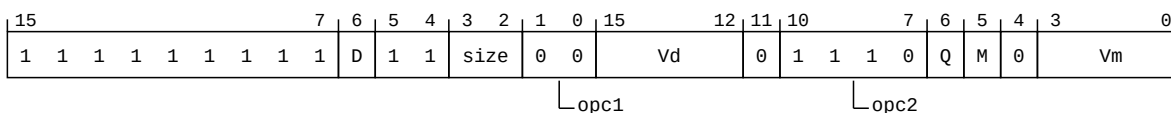
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D:~Vd);
let m : integer = UInt(M:~Vm);
let regs : integer = if O == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

$$VQABS\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \langle Dd \rangle, \quad \langle Dm \rangle$$

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let result : integer = Abs(SInt(D(m+r)[e*:esize]));
            let (res, sat) : (bits(esize), boolean) = SignedSatQ{esize}(result);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

C6.1.153 VQADD

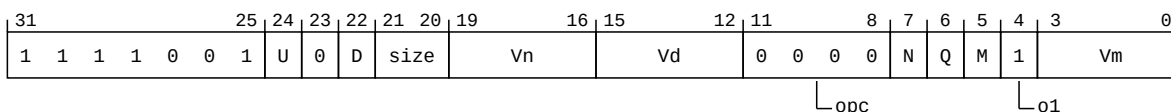
Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, **FPSCR.QC**, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the **CPACR**, **NSACR**, and **HCPTTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

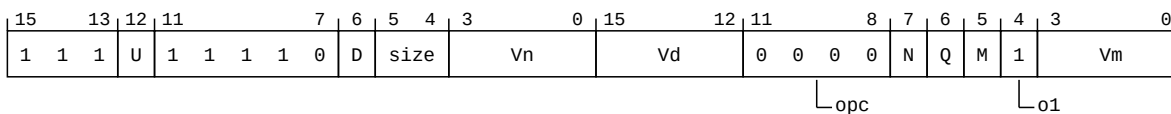
Applies when (Q == 1)

VQADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let sum : integer = element1 + element2;
            let (res, sat) : (bits(esize), boolean) = SatQ{esize}(sum, unsigned);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

C6.1.154 VQDMLAL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector.

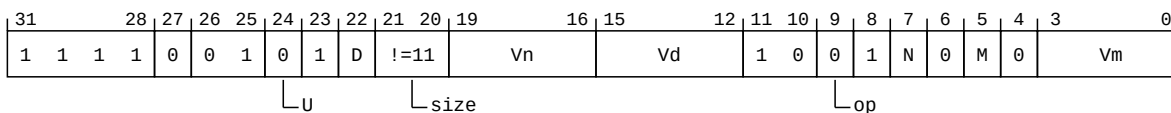
The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, **FPSCR.QC**, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the **CPACR**, **NSACR**, and **HCPTTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

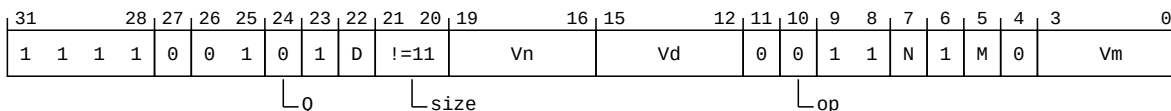
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;

```

A2



Encoding

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

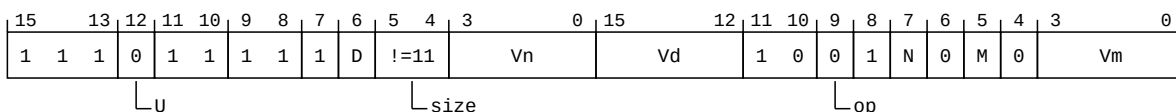
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;

```

T1



Encoding

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

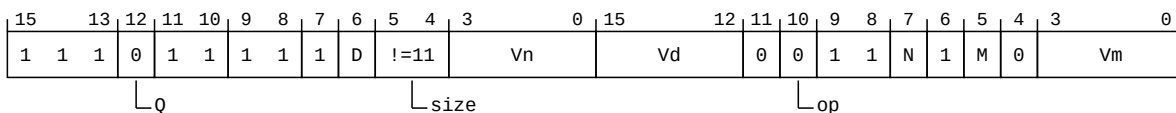
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;

```

T2



Encoding

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” and “A2” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” and “T2” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

For the “A1” and “T1” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.
For the “A2” and “T2” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>” field when <dt> is S16, otherwise the “Vm” field.

<index>

Is the element index in the range 0 to 3, encoded in the “M:Vm<3>” field when <dt> is S16, otherwise in range 0 to 1, encoded in the “M” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(Din(m)[index*:esize]); end;
    for e = 0 to elements-1 do
        if !scalar_form then op2 = SInt(Din(m)[e*:esize]); end;
        let op1 : integer = SInt(Din(n)[e*:esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        let (product, sat1) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);
        var result : integer;
        if add then
            result = SInt(Qin(d>>1)[e*:(2*esize)]) + SInt(product);
        else
            result = SInt(Qin(d>>1)[e*:(2*esize)]) - SInt(product);
        end;
        var sat2 : boolean;
        var res : bits(2*esize);
        (res, sat2) = SignedSatQ{2*esize}(result);
        Q(d>>1)[e*:(2*esize)] = res;
        if sat1 || sat2 then FPSCR().QC = '1'; end;
    end;
end;

```

C6.1.155 VQDMLSL

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

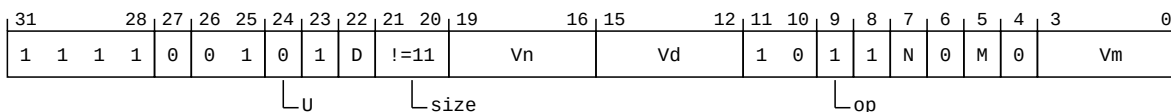
The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

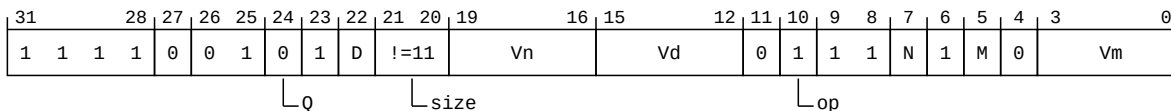
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;

```

A2



Encoding

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

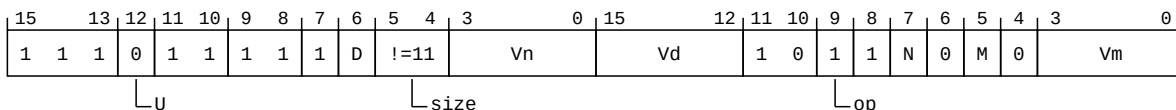
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;

```

T1



Encoding

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

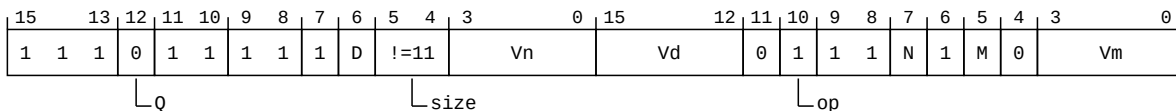
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;

```

T2



Encoding

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let add : boolean = (op == '0');
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” and “A2” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” and “T2” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

For the “A1” and “T1” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.
For the “A2” and “T2” variants: is the 64-bit name of the second SIMD&FP source register, encoded in the “Vm<2:0>” field when <dt> is S16, otherwise the “Vm” field.

<index>

Is the element index in the range 0 to 3, encoded in the “M:Vm<3>” field when <dt> is S16, otherwise in range 0 to 1, encoded in the “M” field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(Din(m)[index*:esize]); end;
    for e = 0 to elements-1 do
        if !scalar_form then op2 = SInt(Din(m)[e*:esize]); end;
        let op1 : integer = SInt(Din(n)[e*:esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        let (product, sat1) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);
        var result : integer;
        if add then
            result = SInt(Qin(d>>1)[e*(2*esize)]) + SInt(product);
        else
            result = SInt(Qin(d>>1)[e*(2*esize)]) - SInt(product);
        end;
        let (res, sat2) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(result);
        Q(d>>1)[e*(2*esize)] = res;
        if sat1 || sat2 then FPSCR().QC = '1'; end;
    end;
end;

```

C6.1.156 VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see [VQRDMULH](#).

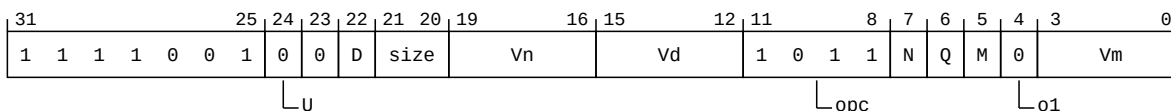
The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

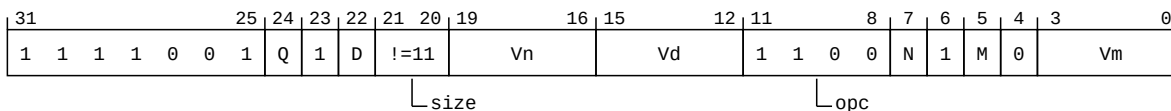
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;

```

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VQDMULH {<c>} {<q>}.<dt> {<Qd>, }<Qn>, <Dm[x]>

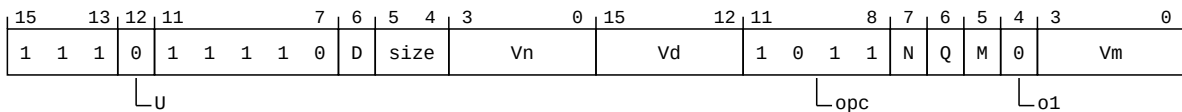
Decode for all variants of this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VQDMULH {<c>} {<q>} .<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

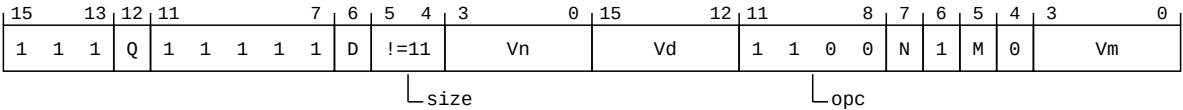
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;

```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in “Vm<2:0>”, and x is encoded in “M:Vm<3>”. If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in “Vm”, and x is encoded in “M”.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    if scalar_form then op2 = SInt(D(m)[index*:esize]); end;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if !scalar_form then op2 = SInt(D(m+r)[e*:esize]); end;
            let op1 : integer = SInt(D(n+r)[e*:esize]);
            // The following only saturates if both op1 and op2 equal -(2^(esize-1))
            let (result, sat) : (bits(esize), boolean) = SignedSatQ(esize)((2*op1*op2) >> esize);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
end;

```

C6.1.157 VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector.

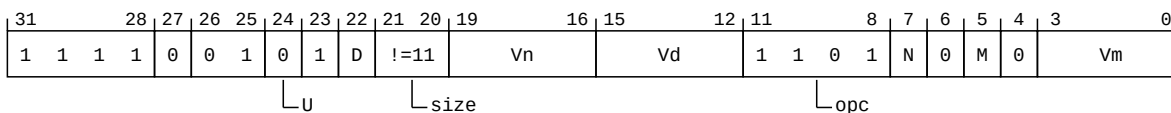
The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, **FPSCR.QC**, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

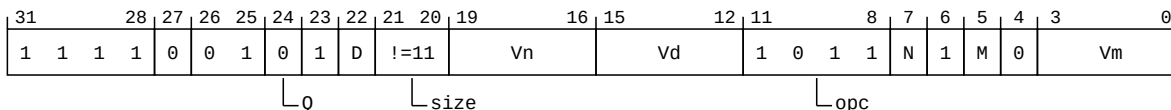
Decode for this encoding

```

if size == '1l' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;

```

A2



Encoding

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

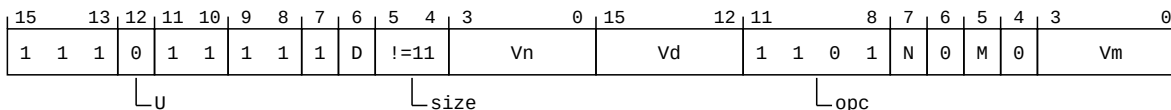
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;

```

T1



Encoding

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

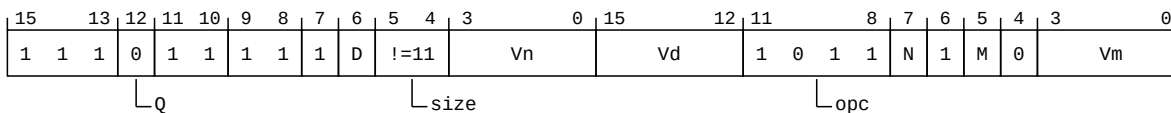
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = FALSE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let index : integer = ARBITRARY : integer;

```

T2



Encoding

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

Decode for this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' || Vd[0] == '1' then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>
For the “A1” and “A2” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” and “T2” variants: see [Standard assembler syntax fields](#).

<q>
See [Standard assembler syntax fields](#).

<dt>
Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd>
Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>
Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>
Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Dm[x]>
Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in “Vm<2:0>”, and x is encoded in “M:Vm<3>”. If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in “Vm”, and x is encoded in “M”.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  var op2 : integer;
  if scalar_form then op2 = SInt(Din(m)[index*esize]); end;
  for e = 0 to elements-1 do
    if !scalar_form then op2 = SInt(Din(m)[e*esize]); end;
```

```
    let op1 : integer = SInt(Din(n)[e*:esize]);  
    // The following only saturates if both op1 and op2 equal  $-(2^{(esize-1)})$   
    let (product, sat) : (bits(2*esize), boolean) = SignedSatQ{2*esize}(2*op1*op2);  
    Q(d>>1)[e*:(2*esize)] = product;  
    if sat then FPSCR().QC = '1'; end;  
end;  
end;
```

C6.1.158 VQMOVN, VQMOVUN

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector.

The operand is a quadword vector. The elements can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

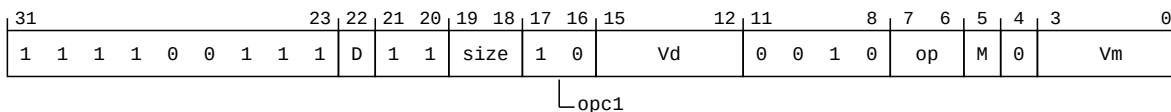
If any of the results overflow, they are saturated. The cumulative saturation bit, **FPSCR.QC**, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instructions `VQRSHRN (zero)`, `VQRSHRUN (zero)`, `VQSHRN (zero)`, and `VQSHRUN (zero)`.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Signed result variant

Applies when (op == 1x)

VQMOVN{<c>} {<q>} .<dt> <Dd>, <Qm>

Encoding for the Unsigned result variant

Applies when (op == 01)

VQMOVUN{<c>} {<q>}.<dt> <Dd>, <Qm>

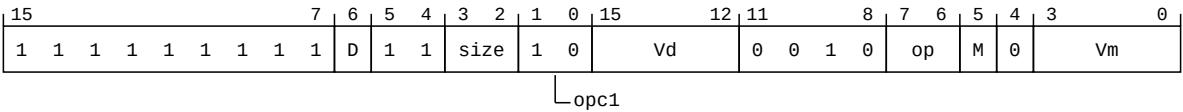
Decode for all variants of this encoding

```

if op == '00' then See("VMOVN"); end;
if size == '11' || Vm[0] == '1' then Undefined(); end;
let src_unsigned : boolean = (op == '11');
let dest_unsigned : boolean = (op[0] == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1



Encoding for the Signed result variant

Applies when (op == 1x)

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Encoding for the Unsigned result variant

Applies when (op == 01)

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for all variants of this encoding

```
if op == '00' then See("VMOVN"); end;  
if size == '11' || Vm[0] == '1' then Undefined(); end;  
let src_unsigned : boolean = (op == '11');  
let dest_unsigned : boolean = (op[0] == '1');  
let esize : integer{} = 8 << UInt(size);  
let elements : integer = 64 DIV esize;  
let d : integer = UInt(D::Vd);  
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the “A1 Signed result” and “A1 Unsigned result” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Signed result” and “T1 Unsigned result” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

For the “A1 Signed result” and “T1 Signed result” variants: is the data type for the elements of the operand, encoded in “op:size”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16

op	size	<dt>
11	01	U32
11	10	U64

For the “A1 Unsigned result” and “T1 Unsigned result” variants: is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let opelt : bits(2*esize) = Qin(m>>1)[e:(2*esize)];
        let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
        let (res, sat) : (bits(esize), boolean) = SatQ{esize}(element, dest_unsigned);
        D(d)[e*:esize] = res;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```

C6.1.159 VQNEG

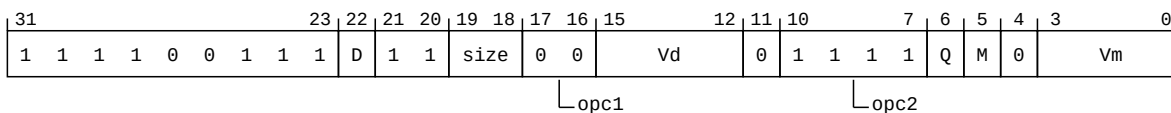
Vector Saturating Negate negates each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, **FPSCR.QC**, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

VONEG{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VQNEG}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \langle Qd \rangle, \langle Qm \rangle$$

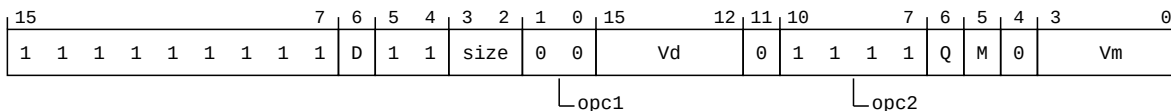
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D:~Vd);
let m : integer = UInt(M:~Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let result : integer = -SInt(D(m+r)[e*:esize]);
            let (res, sat) : (bits(esize), boolean) = SignedSatQ{esize}(result);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

C6.1.160 VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded. For truncated results see [VQDMULH](#).

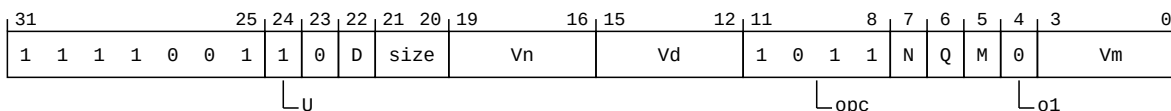
The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

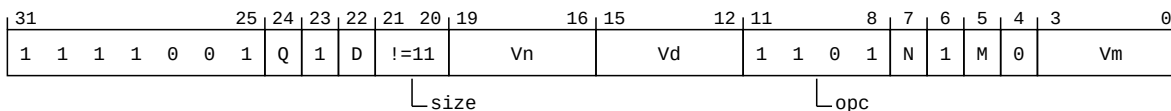
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;

```

A2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x]>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x]>

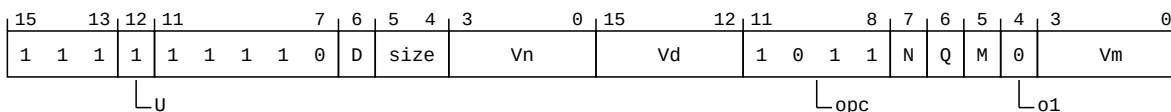
Decode for all variants of this encoding

```

if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

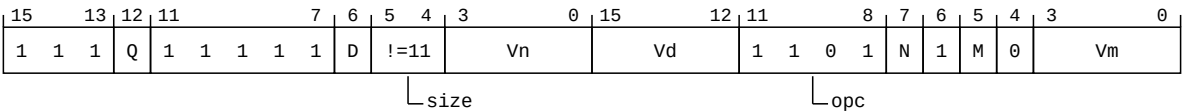
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '00' || size == '11' then Undefined(); end;
let scalar_form : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
let index : integer = ARBITRARY : integer;

```

T2



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x>

Decode for all variants of this encoding

```
if size == '11' then See("Related encodings"); end;
if size == '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1') then Undefined(); end;
let scalar_form : boolean = TRUE;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = if size == '01' then UInt(Vm[2:0]) else UInt(Vm);
let index : integer = if size == '01' then UInt(M::Vm[3]) else UInt(M);
let regs : integer = if Q == '0' then 1 else 2;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
```

Related encodings: See Advanced SIMD data-processing for the T32 instruction set, or Advanced SIMD data-processing for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector”, “A1 64-bit SIMD vector”, “A2 128-bit SIMD vector”, and “A2 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 128-bit SIMD vector”, and “T2 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Dm[x]>

Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in “Vm<2:0>”, and x is encoded in “M:Vm<3>”. If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in “Vm”, and x is encoded in “M”.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var op2 : integer;
    let round : boolean = TRUE;
    if scalar_form then op2 = SInt(D(m)[index*:esize]); end;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1 : integer = SInt(D(n+r)[e*:esize]);
            if !scalar_form then op2 = SInt(D(m+r)[e*:esize]); end;
            let rdmulh : integer = RShr(2*op1*op2, esize, round);
            let (result, sat) : (bits(esize), boolean) = SignedSatQ{esize}(rdmulh);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
end;

```

C6.1.161 VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see [VQSHL \(register\)](#).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

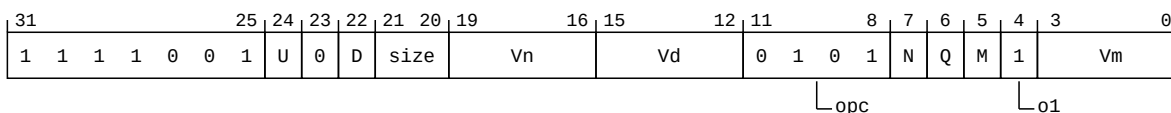
The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQRSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQRSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

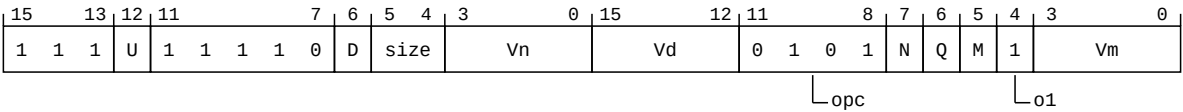
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VQRSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16

U	size	<dt>
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : bits(esize);
    var sat : boolean;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            var element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            if shift >= 0 then // left shift
                element = element << shift;
            else // rounding right shift
                shift = -shift;
                element = (element + (1 << (shift - 1))) >> shift;
            end;
            (result, sat) = SatQ(esize)(element, unsigned);
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;

```

C6.1.162 VQRSHRN, VQRSHRUN

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see [VQSHRN](#) and [VQSHRUN](#).

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

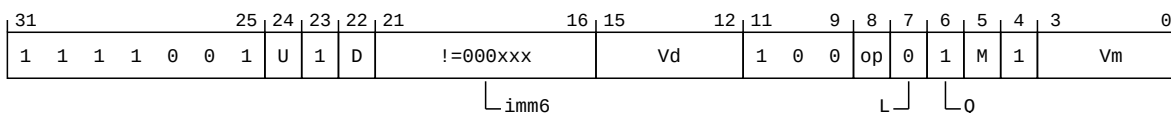
The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Signed result variant

Applies when (op == 1)

VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Encoding for the Unsigned result variant

Applies when (U == 1 && op == 0)

VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

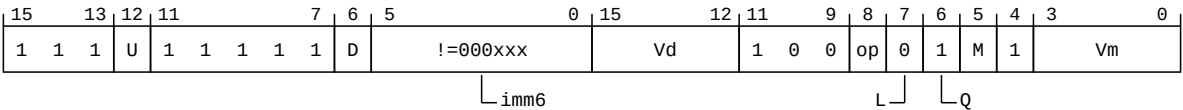
Decode for all variants of this encoding

```

if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VQRSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1



Encoding for the Signed result variant

Applies when (op == 1)

```
VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Encoding for the Unsigned result variant

Applies when (U == 1 && op == 0)

```
VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VQRSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Signed result” and “A1 Unsigned result” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Signed result” and “T1 Unsigned result” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<type>

For the “A1 Signed result” and “T1 Signed result” variants: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S

1	U
---	---

For the “A1 Unsigned result” and “T1 Unsigned result” variants: is S when “U” is 1, and is encoded in “U”.

<size>

Is the data size for the elements of the vectors, encoded in “imm6”:

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<imm>

Is an immediate value, in the range 1 to <size>/2, encoded in the “imm6” field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let opelt : bits(2*esize) = Qin(m>>1)[e*:(2*esize)];
        let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
        let rshrn : integer = RShr(element, shift_amount, round);
        let (result, sat) : (bits(esize), boolean) = SatQ(esize)(rshrn, dest_unsigned);
        D(d)[e*:(esize)] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```


C6.1.163 VQRSHRN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed rounded results in a doubleword vector.

This is a pseudo-instruction of [VQMOVN](#), [VQMOVUN](#). This means:

- The encodings in this description are named to match the encodings of [VQMOVN](#), [VQMOVUN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VQMOVN](#), [VQMOVUN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	D	1	1	size	1	0	Vd	0	0	1	0	1	x	M	0	Vm	
└─opc1														└─op										

Encoding for the Signed result variant

VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	8	7	6	5	4	3	0	
1	1	1	1	1	1	1	1	D	1	1	size	1	0		Vd		0	0	1	0	1	x	M	0		Vm
└opc1														└op												

Encoding for the Signed result variant

VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

<c>

For the “A1 Signed result” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Signed result” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “op:size”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16
11	01	U32
11	10	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VQMOVN](#), [VQMOVUN](#) gives the operational pseudocode for this instruction.

C6.1.164 VQRSHRUN (zero)

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned rounded results in a doubleword vector.

This is a pseudo-instruction of [VQMOVN](#), [VQMOVUN](#). This means:

- The encodings in this description are named to match the encodings of [VQMOVN](#), [VQMOVUN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VQMOVN](#), [VQMOVUN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	1	M	0	Vm	
└─opc1														└─op										

Encoding for the Unsigned result variant

VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	8	7	6	5	4	3	0	
1	1	1	1	1	1	1	1	D	1	1	size	1	0		Vd		0	0	1	0	0	1	M	0		Vm
└opc1														└op												

Encoding for the Unsigned result variant

VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

<c>

For the “A1 Unsigned result” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Unsigned result” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VQMOVN](#), [VQMOVUN](#) gives the operational pseudocode for this instruction.

C6.1.165 VQSHL, VQSHLU (immediate)

Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operand elements must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31						25	24	23	22	21						16	15						12	11						9	8	7	6	5	4	3						0
1	1	1	1	0	0	1	U	1	D	imm6					Vd					0					1	1	op	L	Q	M	1	Vm										

Encoding for the 64-bit SIMD vector, signed result variant

Applies when (op == 1 && Q == 0 && !(imm6[5:3] == 000 && L == 0))

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>
```

Encoding for the 64-bit SIMD vector, unsigned result variant

Applies when (U == 1 && op == 0 && Q == 0 && !(imm6[5:3] == 000 && L == 0))

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector, signed result variant

Applies when (op == 1 && Q == 1 && !(imm6[5:3] == 000 && L == 0))

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>
```

Encoding for the 128-bit SIMD vector, unsigned result variant

Applies when (U == 1 && op == 0 && Q == 1 && !(imm6[5:3] == 000 && L == 0))

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1

15	13	12	11	7	6	5	0				15	12	11	9	8	7	6	5	4	3	0					
1	1	1	U	1	1	1	1	1	D	imm6				Vd		0		1	1	op	L	Q	M	1	Vm	

Encoding for the 64-bit SIMD vector, signed result variant

Applies when (op == 1 && Q == 0 && !(imm6[5:3] == 000 && L == 0))

VQSHL{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 64-bit SIMD vector, unsigned result variant

Applies when (U == 1 && op == 0 && Q == 0 && !(imm6[5:3] == 000 && L == 0))

VQSHLU{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector, signed result variant

Applies when (op == 1 && Q == 1 && !(imm6[5:3] == 000 && L == 0))

VQSHL{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Encoding for the 128-bit SIMD vector, unsigned result variant

Applies when (U == 1 && op == 0 && Q == 1 && !(imm6[5:3] == 000 && L == 0))

VQSHLU{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector, signed result”, “A1 128-bit SIMD vector, unsigned result”, “A1 64-bit SIMD vector, signed result”, and “A1 64-bit SIMD vector, unsigned result” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector, signed result”, “T1 128-bit SIMD vector, unsigned result”, “T1 64-bit SIMD vector, signed result”, and “T1 64-bit SIMD vector, unsigned result” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxx	32
1	xxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 0 to <size>-1, encoded in the “imm6” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
            let (result, sat) : (bits(esize), boolean) = SatQ(esize)(element << shift_amount,
                                                                    dest_unsigned);

            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;
```

C6.1.166 VQSHL (register)

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see [VQRSHL](#).

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

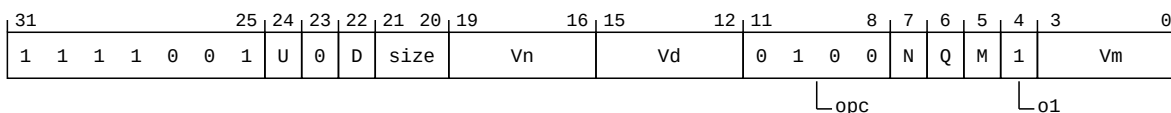
The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

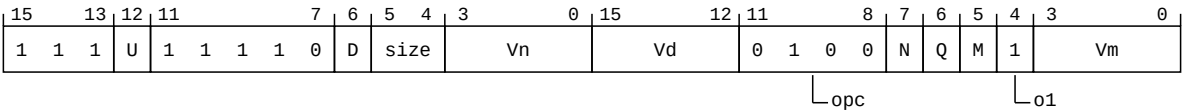
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VQSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VQSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16

U	size	<dt>
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var sat : boolean;
            var result : bits(esize);
            if shift >= 0 then
                (result,sat) = SatQ{esize}(element << shift, unsigned);
            else
                (result,sat) = SatQ{esize}(element >> -shift, unsigned);
            end;
            D(d+r)[e*:esize] = result;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;

```

C6.1.167 VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector.

For rounded results, see [VQRSHRN](#) and [VQRSHRUN](#).

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

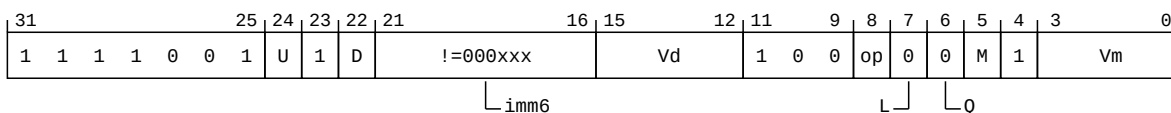
The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Signed result variant

Applies when (op == 1)

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Encoding for the Unsigned result variant

Applies when (U == 1 && op == 0)

VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

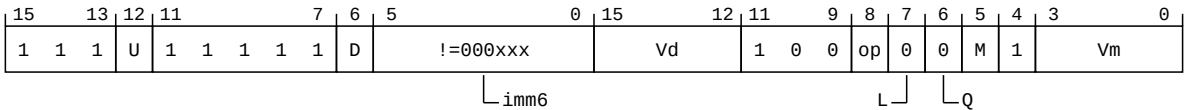
Decode for all variants of this encoding

```

if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

T1



Encoding for the Signed result variant

Applies when (op == 1)

```
VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Encoding for the Unsigned result variant

Applies when (U == 1 && op == 0)

```
VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if U == '0' && op == '0' then See("VSHRN"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let src_unsigned : boolean = (U == '1' && op == '1');
let dest_unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Signed result” and “A1 Unsigned result” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Signed result” and “T1 Unsigned result” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<type>

For the “A1 Signed result” and “T1 Signed result” variants: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S

1	U
---	---

For the “A1 Unsigned result” and “T1 Unsigned result” variants: is S when “U” is 1, and is encoded in “U”.

<size>

Is the data size for the elements of the vectors, encoded in “imm6”:

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<imm>

Is an immediate value, in the range 1 to <size>/2, encoded in the “imm6” field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let opelt : bits(2*esize) = Qin(m>>1)[e*:(2*esize)];
        let element : integer = if src_unsigned then UInt(opelt) else SInt(opelt);
        let (result, sat) : (bits(esize), boolean) = SatQ(esize)(element >> shift_amount,
                                                                    dest_unsigned);

        D(d)[e*esize] = result;
        if sat then FPSCR().QC = '1'; end;
    end;
end;
```

C6.1.168 VQSHRN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed truncated results in a doubleword vector.

This is a pseudo-instruction of [VQMOVN](#), [VQMOVUN](#). This means:

- The encodings in this description are named to match the encodings of [VQMOVN](#), [VQMOVUN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VQMOVN](#), [VQMOVUN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	D	1	1	size	1	0	Vd	0	0	1	0	1	x	M	0	Vm	
														</										

<dt>

Is the data type for the elements of the operand, encoded in “op:size”:

op	size	<dt>
10	00	S16
10	01	S32
10	10	S64
1x	11	RESERVED
11	00	U16
11	01	U32
11	10	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VQMOVN](#), [VQMOVUN](#) gives the operational pseudocode for this instruction.

C6.1.169 VQSHRUN (zero)

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned truncated results in a doubleword vector.

This is a pseudo-instruction of [VQMOVN](#), [VQMOVUN](#). This means:

- The encodings in this description are named to match the encodings of [VQMOVN](#), [VQMOVUN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VQMOVN](#), [VQMOVUN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	8	7	6	5	4	3	0
1	1	1	1	0	0	1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	1	M	0	Vm	
└─opc1														└─op										

Encoding for the Unsigned result variant

VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	8	7	6	5	4	3	0	
1	1	1	1	1	1	1	1	D	1	1	size	1	0		Vd		0	0	1	0	0	1	M	0		Vm
└opc1														└op												

Encoding for the Unsigned result variant

VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

<c>

For the “A1 Unsigned result” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Unsigned result” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VQMOVN](#), [VQMOVUN](#) gives the operational pseudocode for this instruction.

C6.1.170 VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct.

The operand and result elements must all be the same type, and can be any one of:

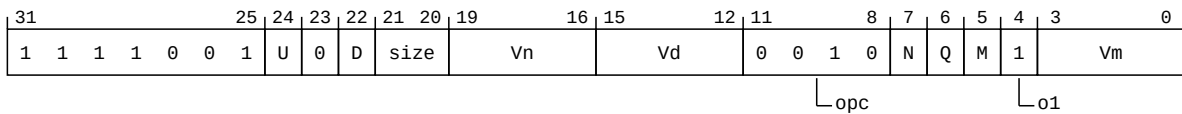
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation bit, **FPSCR.QC**, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the **CPACR**, **NSACR**, and **HCPTTR** registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be **UNDEFINED**, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

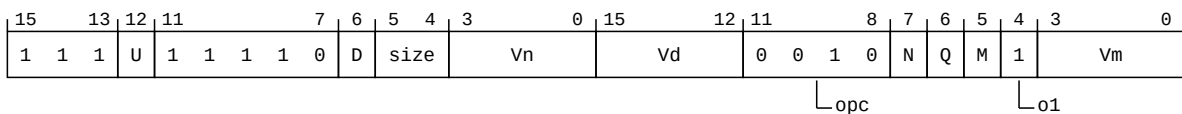
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VQSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VQSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let diff : integer = element1 - element2;
            let (res, sat) : (bits(esize), boolean) = SatQ{esize}(diff, unsigned);
            D(d+r)[e*:esize] = res;
            if sat then FPSCR().QC = '1'; end;
        end;
    end;
end;

```

C6.1.171 VRADDHN

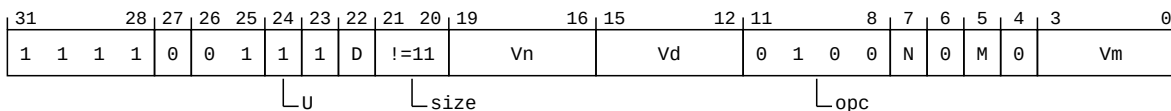
Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. For truncated results, see [VADDHN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

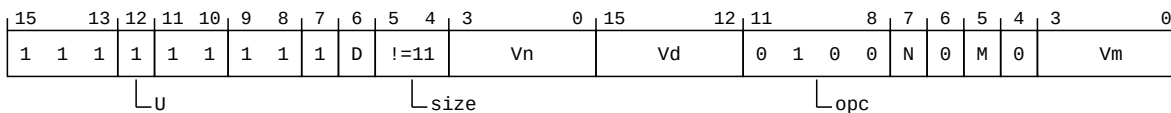
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1



Encoding

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = (RShr(UInt(Qin(n>>1)[e:(2*esize)] +
                                   Qin(m>>1)[e:(2*esize)]), esize, round));
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```

C6.1.172 VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector.

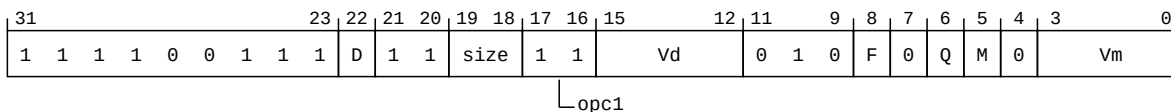
The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see [Floating-point reciprocal square root estimate and step](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>

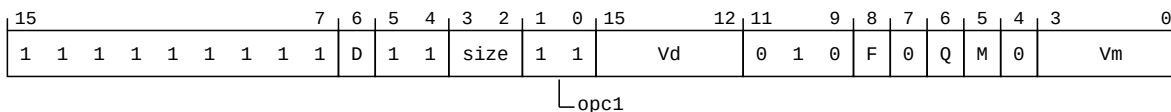
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	10	U32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see [Floating-point reciprocal estimate and step](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if floating_point then
                D(d+r)[e*:size] = FPRecipEstimate{size}(D(m+r)[e*:size], fpcr);
            else
                D(d+r)[e*:size] = UnsignedRecipEstimate{size}(D(m+r)[e*:size]);
            end;
        end;
    end;
end;
```

C6.1.173 VRECPS

Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector.

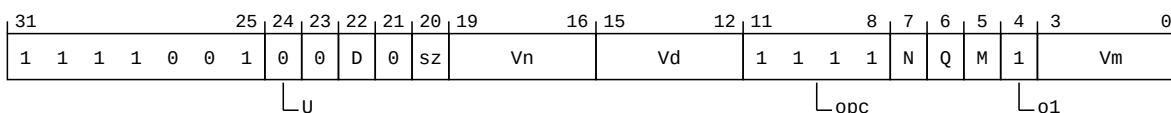
The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see [Floating-point reciprocal estimate and step](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

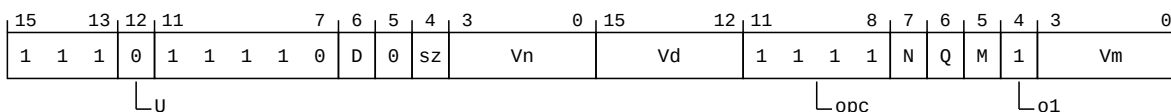
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let esize : integer() = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see [Floating-point reciprocal estimate and step](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = FPREcipStep{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize]);
        end;
    end;
end;
```

C6.1.174 VREV16

Vector Reverse in halfwords reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

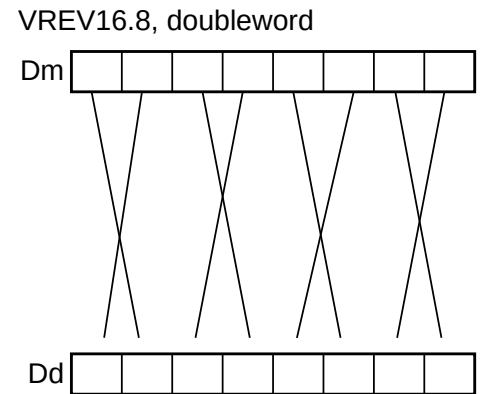
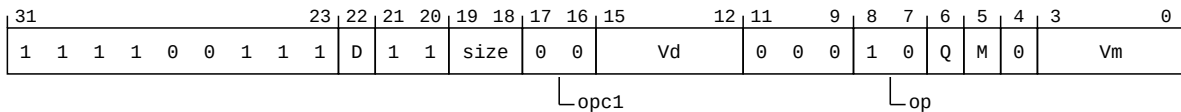


Figure C6-6 VREV16 doubleword operation

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VREV16{<c>}{<q>}.<dt>    <Qd>, <Qm>
```

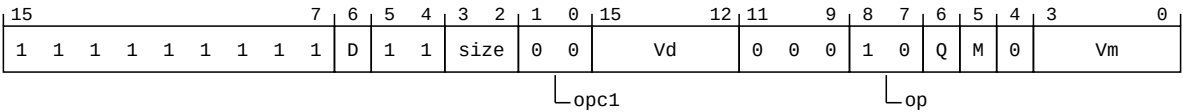
Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
```

```
let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements per container : integer = container_size DIV esize;
```

```
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

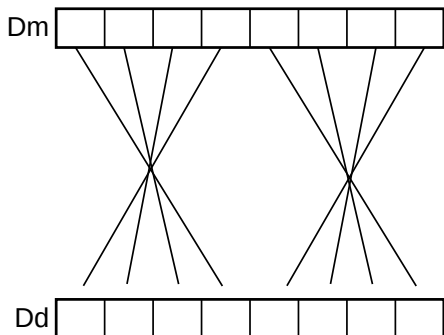
    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*:esize] = D(m+r)[element*:esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
        end;
        D(d+r) = result;
    end;
end;
```

C6.1.175 VREV32

Vector Reverse in words reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

VREV32.8, doubleword



VREV32.16, doubleword

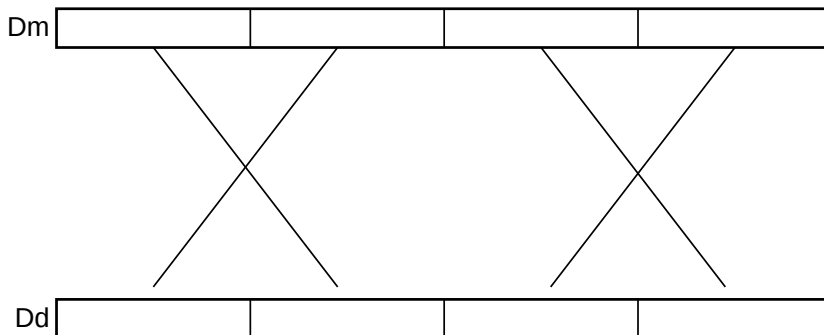
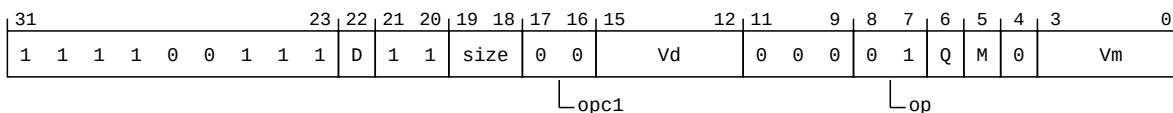


Figure C6-7 VREV32 doubleword operations

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```

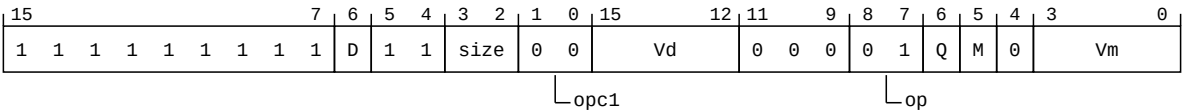
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

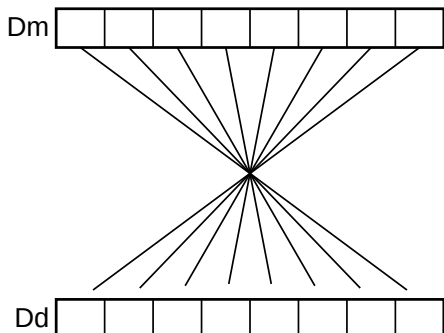
    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*:esize] = D(m+r)[element*:esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
        end;
        D(d+r) = result;
    end;
end;
```

C6.1.176 VREV64

Vector Reverse in doublewords reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

VREV64.8, doubleword



VREV64.32, quadword

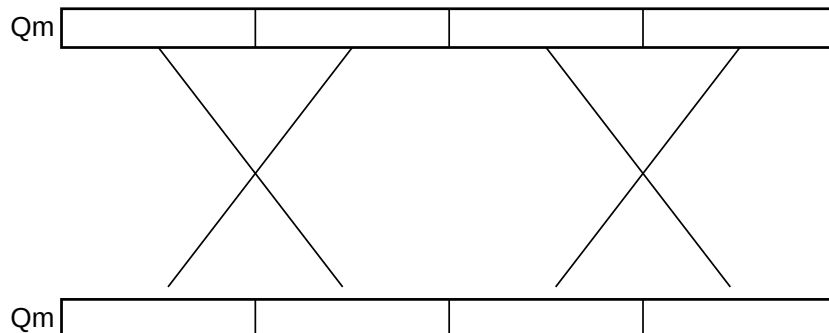
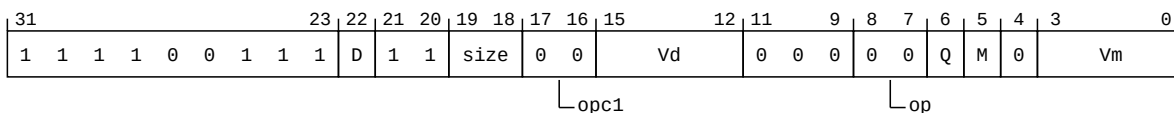


Figure C6-8 VREV64 doubleword operations

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```

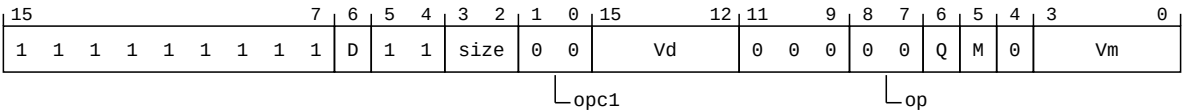
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if UInt(op)+UInt(size) >= 3 then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;

let esize : integer{} = 8 << UInt(size);
let container_size : integer = 64 >> UInt(op);
let containers : integer = 64 DIV container_size;
let elements_per_container : integer = container_size DIV esize;

let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var result : bits(64);
    var element : integer;
    var rev_element : integer;
    for r = 0 to regs-1 do
        element = 0;
        for c = 0 to containers-1 do
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1 do
                result[rev_element*:esize] = D(m+r)[element*:esize];
                element = element + 1;
                rev_element = rev_element - 1;
            end;
        end;
        D(d+r) = result;
    end;
end;
```

C6.1.177 VRHADD

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

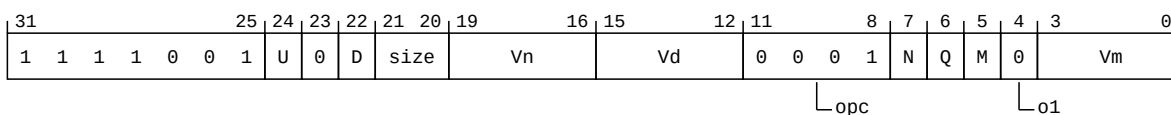
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded. For truncated results, see [VHADD](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

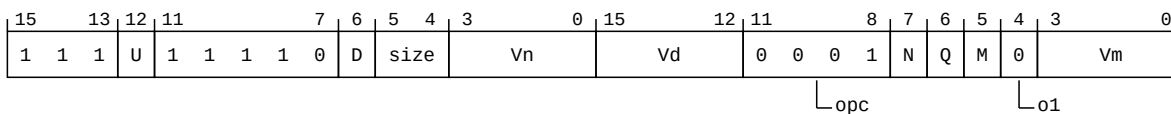
Applies when ($Q == 1$)

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let op1elt : bits(esize) = D(n+r)[e*:esize];
            let op2elt : bits(esize) = D(m+r)[e*:esize];
            let element1 : integer = if unsigned then UInt(op1elt) else SInt(op1elt);
            let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
            let result : integer = (element1 + element2 + 1) >> 1;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

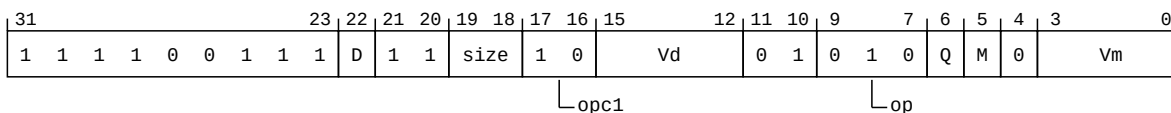
C6.1.178 VRINTA (Advanced SIMD)

Vector Round floating-point to integer towards Nearest with Ties to Away

Vector Round floating-point to integer towards Nearest with Ties to Away rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRINTA{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VRINTA{ <q> } .<dt>    <Od> ,  <Om>
```

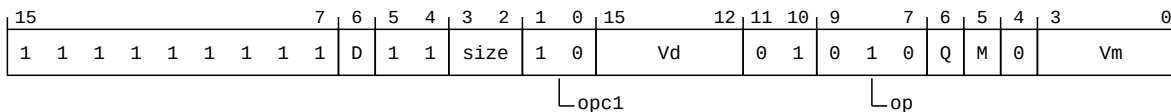
Decode for all variants of this encoding

```

if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRINTA{ <q> } .<dt>    <Dd> ,  <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTA(<q>).<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD two registers misc](#) for the T32 instruction set, or [Advanced SIMD two registers misc](#) for the A32 instruction set.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

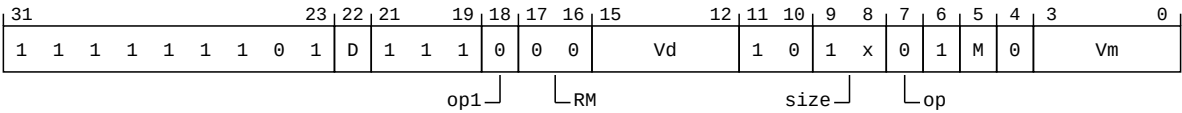
```
EncodingSpecificOperations();  
CheckAdvSIMDEnabled();  
let fpcr : FPCR_Type = StandardFPCR();  
for r = 0 to regs-1 do  
    for e = 0 to elements-1 do  
        let operand1 : bits(usize) = D(m+r)[e*:usize];  
        let result : bits(usize) = FPRoundInt{}(operand1, fpcr, rounding, exact);  
        D(d+r)[e*:usize] = result;  
    end;  
end;
```

C6.1.179 VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTA(<q>).F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTA(<q>).F64 <Dd>, <Dm>

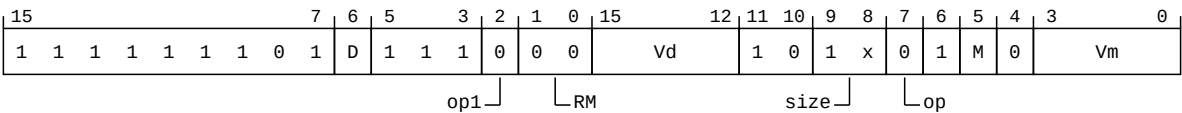
Decode for all variants of this encoding

```

if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VRINTA(<q>).F32 <Sd>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VRINTA(<q>).F64 <Dd>, <Dm>

Decode for all variants of this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 32 =>
        S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
    when 64 =>
        D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;

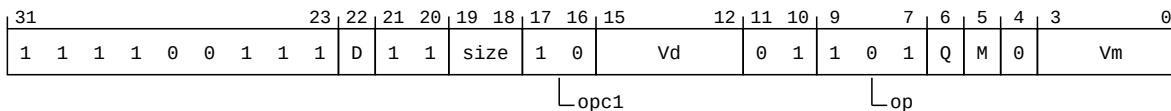
```

C6.1.180 VRINTM (Advanced SIMD)

Vector Round floating-point to integer towards -Infinity rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

```
VRINTM{ <q> } .<dt>    <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VRINTM{ <q> } .<dt>    <Qd>, <Qm>
```

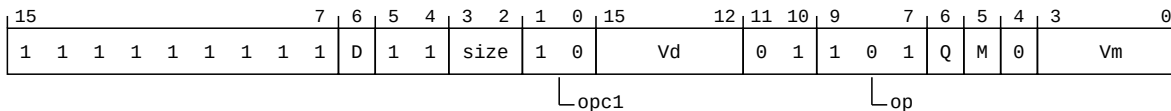
Decode for all variants of this encoding

```

if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRINTM{ <q> } .<dt>    <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTM(<q>).<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD two registers misc](#) for the T32 instruction set, or [Advanced SIMD two registers misc](#) for the A32 instruction set.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

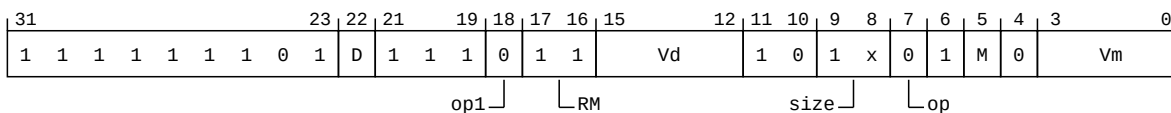
```
EncodingSpecificOperations();  
CheckAdvSIMDEnabled();  
let fpcr : FPCR_Type = StandardFPCR();  
for r = 0 to regs-1 do  
    for e = 0 to elements-1 do  
        let operand1 : bits(usize) = D(m+r)[e*:usize];  
        let result : bits(usize) = FPRoundInt{}(operand1, fpcr, rounding, exact);  
        D(d+r)[e*:usize] = result;  
    end;  
end;  
end;
```

C6.1.181 VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTM(<q>).F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

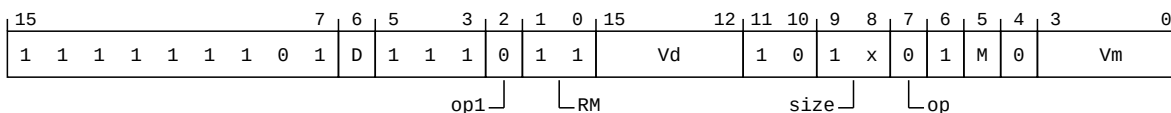
Applies when (size == 11)

```
VRINTM(<q>).F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTM(<q>).F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTM(<q>).F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 32 =>
        S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
    when 64 =>
        D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;

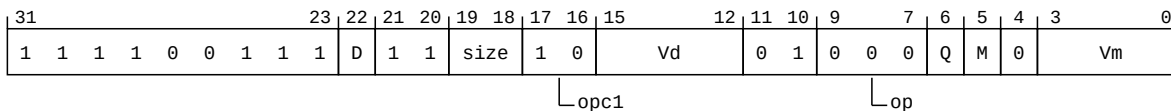
```

C6.1.182 VRINTN (Advanced SIMD)

Vector Round floating-point to integer to Nearest rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

```
VRINTN{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VRINTN{ <q> } .<dt>    <Qd>, <Qm>
```

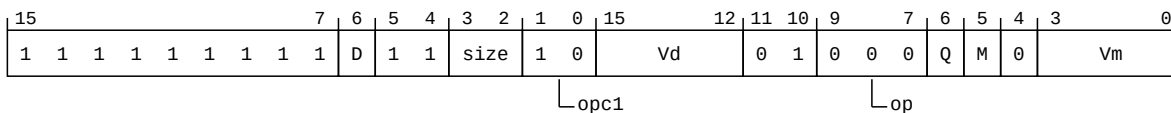
Decode for all variants of this encoding

```

if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRINTN{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTN(<q>).<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD two registers misc](#) for the T32 instruction set, or [Advanced SIMD two registers misc](#) for the A32 instruction set.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

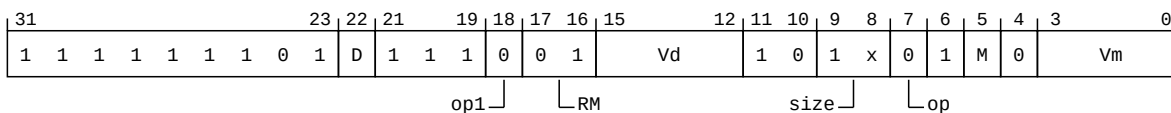
```
EncodingSpecificOperations();  
CheckAdvSIMDEnabled();  
let fpcr : FPCR_Type = StandardFPCR();  
for r = 0 to regs-1 do  
    for e = 0 to elements-1 do  
        let operand1 : bits(usize) = D(m+r)[e*:usize];  
        let result : bits(usize) = FPRoundInt{}(operand1, fpcr, rounding, exact);  
        D(d+r)[e*:usize] = result;  
    end;  
end;  
end;
```

C6.1.183 VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTN(<q>).F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

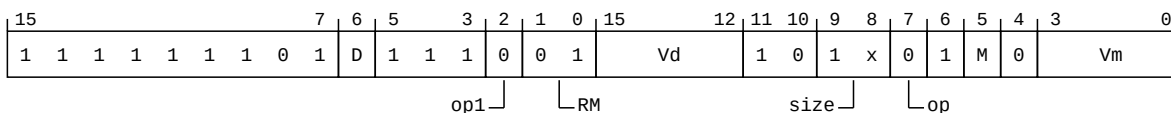
Applies when (size == 11)

```
VRINTN(<q>).F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTN(<q>).F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTN(<q>).F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

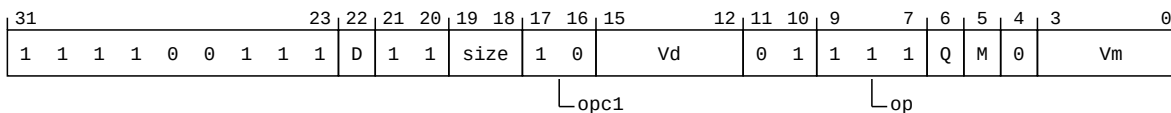
```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
  when 32 =>
    S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
  when 64 =>
    D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;
```


C6.1.184 VRINTP (Advanced SIMD)

Vector Round floating-point to integer towards +Infinity rounds a vector of floating-point values to integral floating-point values of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRINTP{ <q> } . <dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VRINTP{<q>}.<dt> <Qd>, <Qm>
```

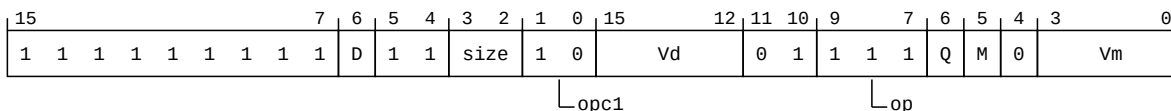
Decode for all variants of this encoding

```

if op[2] != op[0] then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let reqs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when $(Q == 0)$

```
VRINTP{ <q> } . <dt> <Dd> , <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTP(<q>).<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if op[2] != op[0] then See("Related encodings"); end;
if InITBlock() then UnpredictableProcedure(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
// Rounding encoded differently from other VCVT and VRINT instructions
let rounding : FPRounding = FPDecodeRM(op[2]::NOT(op[1]));
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See [Advanced SIMD two registers misc](#) for the T32 instruction set, or [Advanced SIMD two registers misc](#) for the A32 instruction set.

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

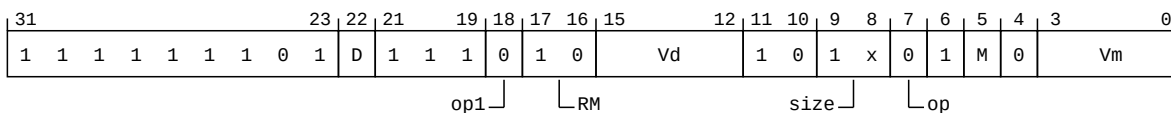
```
EncodingSpecificOperations();  
CheckAdvSIMDEnabled();  
let fpcr : FPCR_Type = StandardFPCR();  
for r = 0 to regs-1 do  
    for e = 0 to elements-1 do  
        let operand1 : bits(esize) = D(m+r)[e*:esize];  
        let result : bits(esize) = FPRoundInt{}(operand1, fpcr, rounding, exact);  
        D(d+r)[e*:esize] = result;  
    end;  
end;  
end;
```

C6.1.185 VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTP{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

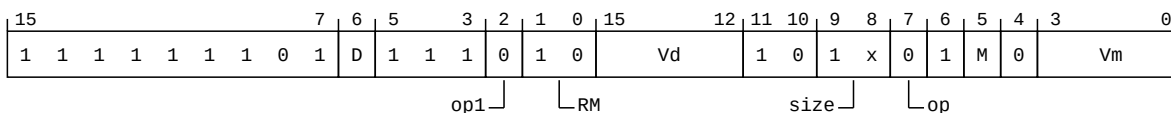
Applies when (size == 11)

```
VRINTP{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTP{<q>}.F32  <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTP{<q>}.F64  <Dd>, <Dm>
```

Decode for all variants of this encoding

```

if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let rounding : FPRounding = FPDecodeRM(RM);
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);

```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```

EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
let fpcr : FPCR_Type = EffectiveFPCR();
case esize of
    when 32 =>
        S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
    when 64 =>
        D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
end;

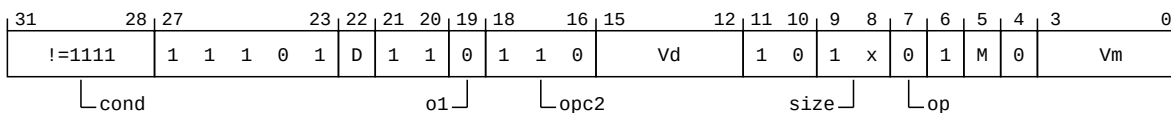
```

C6.1.186 VRINTR

Round floating-point to integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in the FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

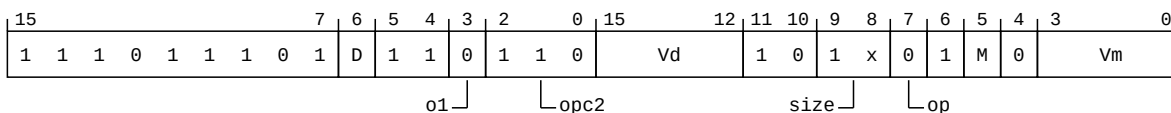
Applies when (size == 11)

```
VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

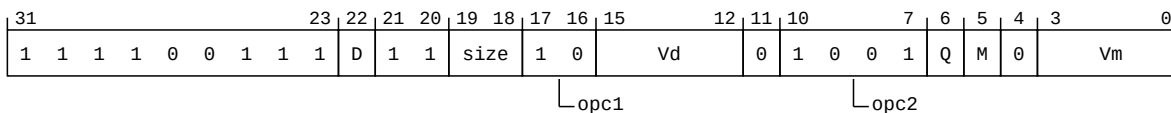
```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    case esize of
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;
```

C6.1.187 VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRINTX{ <q> } .<dt>    <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VRINTX{<q>}.<dt> <Qd>, <Qm>
```

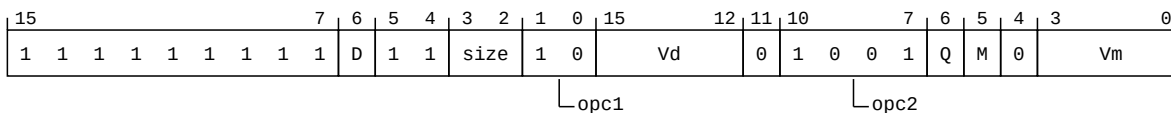
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPRounding_TIEEVEN;
let exact : boolean = TRUE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRINTX{ <q> } . <dt> <Dd>, <Dm>
```


Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRINTX{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPRounding_TIEEVEN;
let exact : boolean = TRUE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

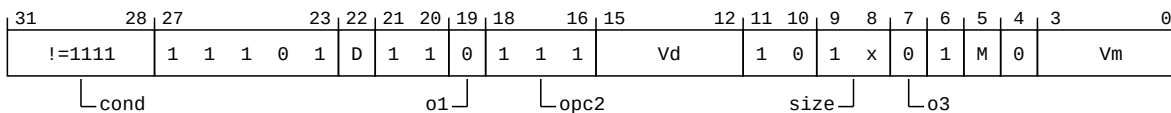
```
EncodingSpecificOperations();  
CheckAdvSIMDEnabled();  
let fpcr : FPCR_Type = StandardFPCR();  
for r = 0 to regs-1 do  
    for e = 0 to elements-1 do  
        let operand1 : bits(usize) = D(m+r)[e*:usize];  
        let result : bits(usize) = FPRoundInt{}(operand1, fpcr, rounding, exact);  
        D(d+r)[e*:usize] = result;  
    end;  
end;  
end;
```

C6.1.188 VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

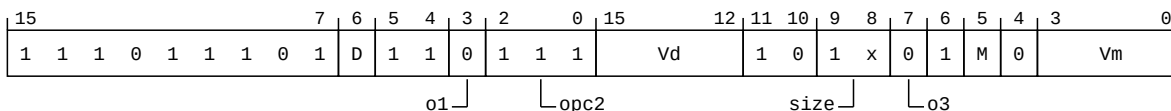
Applies when (size == 11)

```
VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let exact : boolean = TRUE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

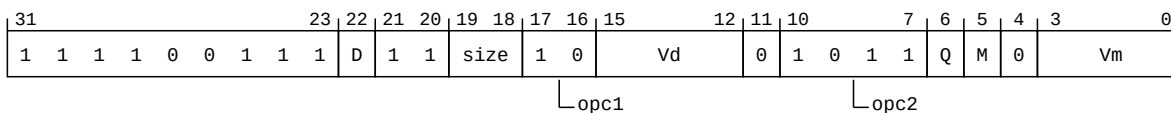
```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = FPRoundingMode(fpcr);
    case esize of
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;
```

C6.1.189 VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

```
VRINTZ{ <q> } . <dt> <Dd> , <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VRINTZ{<q>}.<dt> <Qd>, <Qm>
```

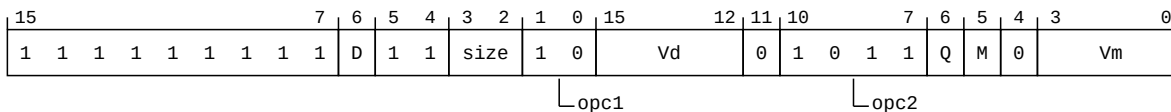
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPRounding_ZERO;
let exact : boolean = FALSE;
let esize : integer{} = 32; let elements : integer{} = 2;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

```
VRINTZ{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

```
VRINTZ{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let rounding : FPRounding = FPRounding_ZERO;
let exact : boolean = FALSE;
let esize : integer{} = 32;
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
if InITBlock() then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

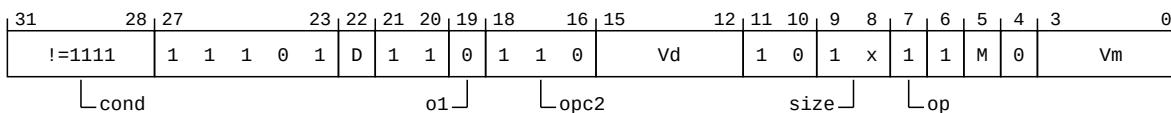
```
EncodingSpecificOperations();  
CheckAdvSIMDEnabled();  
let fpcr : FPCR_Type = StandardFPCR();  
for r = 0 to regs-1 do  
    for e = 0 to elements-1 do  
        let operand1 : bits(usize) = D(m+r)[e*:usize];  
        let result : bits(usize) = FPRoundInt{}(operand1, fpcr, rounding, exact);  
        D(d+r)[e*:usize] = result;  
    end;  
end;  
end;
```

C6.1.190 VRINTZ (floating-point)

Round floating-point to integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

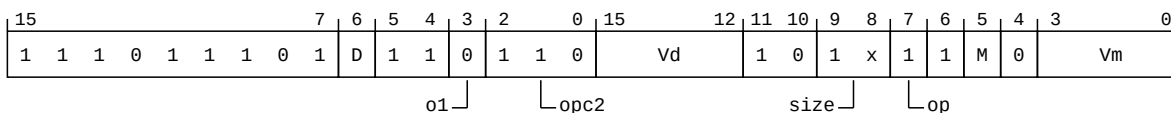
Applies when (size == 11)

```
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let zero_rounding : boolean = (op == '1');
let exact : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    let rounding : FPRounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    case esize of
        when 32 =>
            S(d) = FPRoundInt{32}(S(m), fpcr, rounding, exact);
        when 64 =>
            D(d) = FPRoundInt{64}(D(m), fpcr, rounding, exact);
    end;
end;
```

C6.1.191 VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see VSHL.

The first operand and result elements are the same data type, and can be any one of:

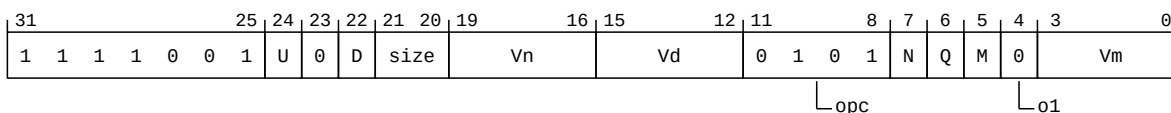
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VRSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>
```

Encoding for the 128-bit SIMD vector variant

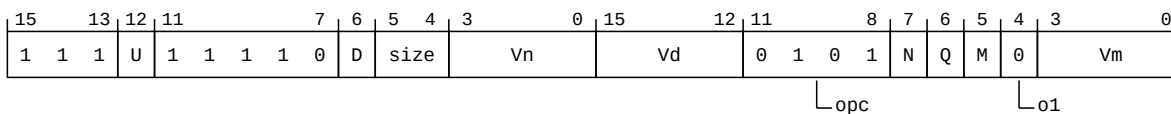
Applies when (Q == 1)

```
VRSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    var result : integer;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            if shift >= 0 then // left shift
                result = element << shift;
            else // rounding right shift
                shift = -shift;
                result = (element + (1 << (shift - 1))) >> shift;
            end;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

C6.1.192 VRSHR

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see [VSHR](#).

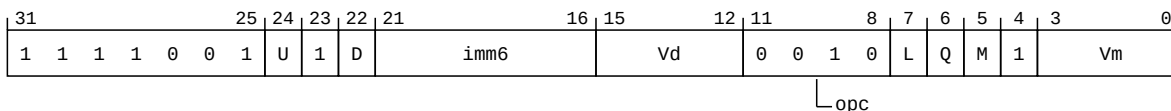
The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

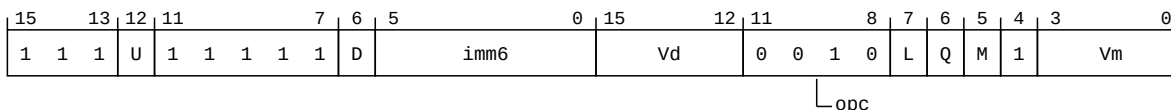
Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VRSHR{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the “imm6” field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = RShr(element, shift_amount, round);
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

C6.1.193 VRSHR (zero)

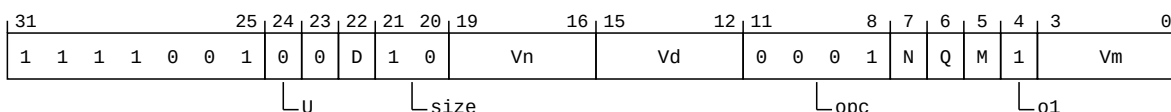
Vector Rounding Shift Right copies the contents of one SIMD register to another.

This is a pseudo-instruction of [VORR \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(register\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VORR \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

`VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0`

is equivalent to

`VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>`

Encoding for the 128-bit SIMD vector variant

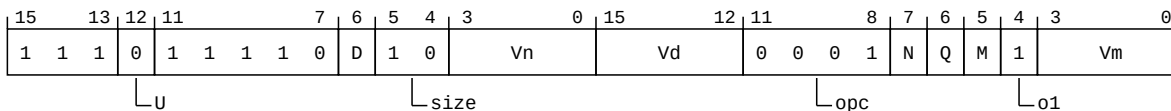
Applies when ($Q == 1$)

`VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0`

is equivalent to

`VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>`

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

`VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0`

is equivalent to

`VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$VRSHR\{<c>\}\{<q>\}.<dt>\ <Qd>, <Qm>, \#0$

is equivalent to

$VORR\{<c>\}\{<q>\}\{.<dt>\}\ \{<Qd>, \}<Qm>, <Qm>$

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “N:Vn” and “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as $<Qd>*2$.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “N:Vn” and “M:Vm” field as $<Qm>*2$.

Operation

The description of [VORR \(register\)](#) gives the operational pseudocode for this instruction.

C6.1.194 VRSHRN

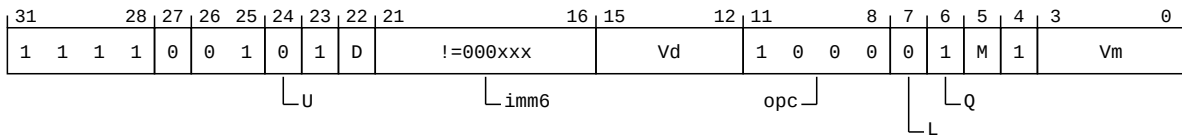
Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see [VSHRN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

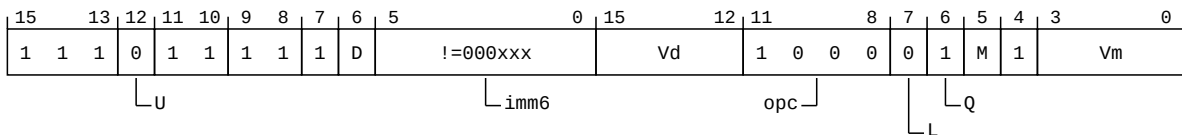
```
VRSHRN(<c>){<q>}.I<size> <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```

if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
  
```

T1



Encoding

```
VRSHRN(<c>){<q>}.I<size> <Dd>, <Qm>, #<imm>
```

Decode for this encoding

```

if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize << 1) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
  
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size for the elements of the vectors, encoded in “imm6”:

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<imm>

Is an immediate value, in the range 1 to <size>/2, encoded in the “imm6” field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = RShr(UInt(Qin(m>>1)[e*(2*esize)]), shift_amount, round);
        D(d)[e*esize] = result[esize-1:0];
    end;
end;
```

C6.1.195 VRSHRN (zero)

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector.

This is a pseudo-instruction of [VMOVN](#). This means:

- The encodings in this description are named to match the encodings of [VMOVN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VMOVN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	7	6	5	4	3	0
1	1	1	1	0	0	1	1	D	1	1	size	1	0	Vd	0	0	1	0	0	0	M	0	Vm
opc1														opc2				Q					

Encoding for the A1 variant

VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	12	11	10	7	6	5	4	3	0	
1	1	1	1	1	1	1	1	D	1	1	size	1	0		Vd		0	0	1	0	0	0	M	0	Vm
└opc1														opc2┐				└0							

Encoding for the T1 variant

VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I 16
01	I 32
10	I 64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VMOVN](#) gives the operational pseudocode for this instruction.

C6.1.196 VRSQRTE

Vector Reciprocal Square Root Estimate finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

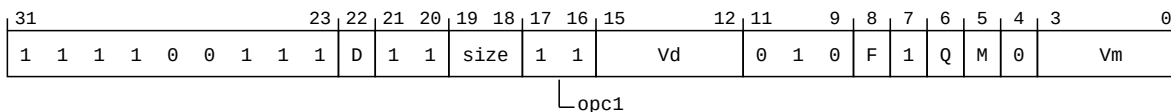
The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see [Floating-point reciprocal estimate and step](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

VRSORTE{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VRSORTE{<c>}{<q>}.<dt> <Qd>, <Qm>

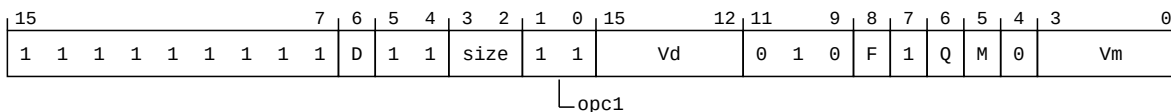
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

VRSORTE{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size != '10' then Undefined(); end;
let floating_point : boolean = (F == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	10	U32
1	01	F16
1	10	F32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see [Floating-point reciprocal estimate and step](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let fpcr : FPCR_Type = StandardFPCR();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if floating_point then
                D(d+r)[e*:size] = FPRSqrtEstimate{size}(D(m+r)[e*:size], fpcr);
            else
                D(d+r)[e*:size] = UnsignedRSqrtEstimate{size}(D(m+r)[e*:size]);
            end;
        end;
    end;
end;
```

C6.1.197 VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.

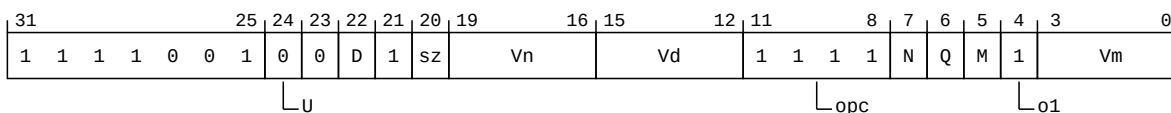
The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see [Floating-point reciprocal estimate and step](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

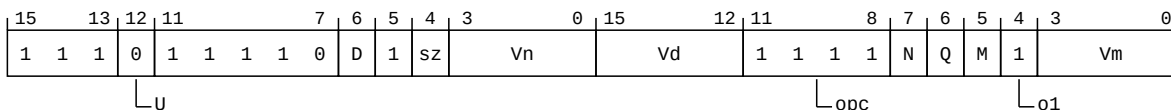
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Newton-Raphson iteration

For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see [Floating-point reciprocal estimate and step](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = FPRSqrtStep{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize]);
        end;
    end;
end;
```

C6.1.198 VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector. For truncated results, see [VSRA](#).

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	25	24	23	22	21	16	15	12	11	8	7	6	5	4	3	0									
1	1	1	1	0	0	1	U	1	D	imm6			Vd			0	0	1	1	L	Q	M	1	Vm	
											└─ opc														

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VRSRA{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VRSRA{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```

if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1

15	13	12	11	7	6	5	0	15	12	11	8	7	6	5	4	3	0								
1	1	1	U	1	1	1	1	1	D	imm6				Vd		0	0	1	1	L	Q	M	1	Vm	
															└─ opc										

Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VRSRA{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VRSRA{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the “imm6” field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = RShr(element, shift_amount, round);
            D(d+r)[e*:esize] = D(d+r)[e*:esize] + result;
        end;
    end;
end;
```

C6.1.199 VRSUBHN

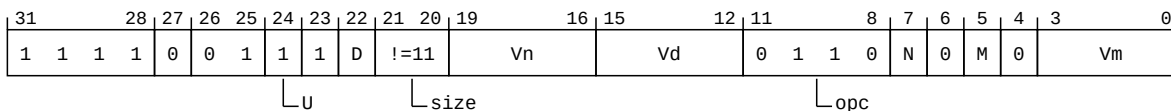
Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. For truncated results, see [VSUBHN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

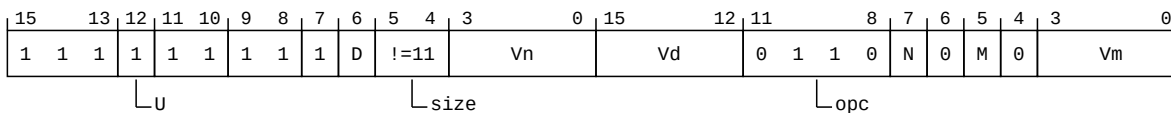
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1



Encoding

VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

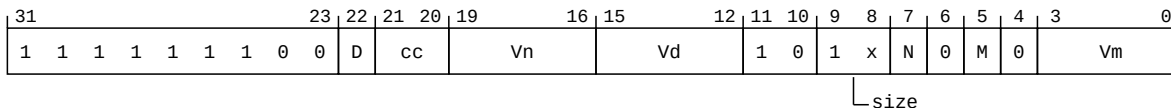
```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let round : boolean = TRUE;
    for e = 0 to elements-1 do
        let result : integer = (RShr(UInt(Qin(n>>1)[e:(2*esize)] -
                                   Qin(m>>1)[e:(2*esize)]), esize, round));
        D(d)[e*:esize] = result[esize-1:0];
    end;
end;
```


C6.1.200 VSELEQ, VSELGE, VSELGT, VSELVS

Floating-point conditional select allows the destination register to take the value in either one or the other source register according to the condition codes in the [APSR](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Equal, single-precision scalar variant

Applies when (cc == 00 && size == 10)

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Equal, double-precision scalar variant

Applies when (cc == 00 && size == 11)

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Encoding for the Greater than or Equal, single-precision scalar variant

Applies when `(cc == 10 && size == 10)`

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Greater than or Equal, double-precision scalar variant

Applies when `(cc == 10 && size == 11)`

```
VSELGE.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Encoding for the Greater than, single-precision scalar variant

Applies when (cc == 11 && size == 10)

```
VSELGT.F32  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Greater than, double-precision scalar variant

Applies when `(cc == 11 && size == 11)`

```
VSELGT.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Encoding for the Unordered, single-precision scalar variant

Applies when (cc == 01 && size == 10)

```
VSELVS.F32  <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Encoding for the Unordered, double-precision scalar variant

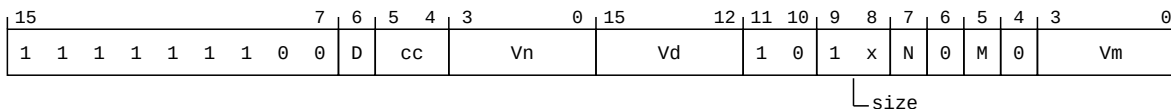
Applies when (cc == 01 && size == 11)

```
VSELVS.F64  <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let condition : bits(4) = cc::(cc[1] XOR cc[0])::'0';
```

T1



Encoding for the Equal, single-precision scalar variant

Applies when (cc == 00 && size == 10)

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Equal, double-precision scalar variant

Applies when (cc == 00 && size == 11)

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Encoding for the Greater than or Equal, single-precision scalar variant

Applies when (cc == 10 && size == 10)

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Greater than or Equal, double-precision scalar variant

Applies when (cc == 10 && size == 11)

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Encoding for the Greater than, single-precision scalar variant

Applies when (cc == 11 && size == 10)

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Greater than, double-precision scalar variant

Applies when (cc == 11 && size == 11)

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Encoding for the Unordered, single-precision scalar variant

Applies when (cc == 01 && size == 10)

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Encoding for the Unordered, double-precision scalar variant

Applies when (cc == 01 && size == 11)

VSELVS.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

Decode for all variants of this encoding

```
if InITBlock() then UnpredictableProcedure(); end;
if size != '1x' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let condition : bits(4) = cc::(cc[1] XOR cc[0])::'0';
```

CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
EncodingSpecificOperations();
CheckVFPEEnabled(TRUE);
case esize of
  when 32 =>
    S(d) = if ConditionHolds(condition) then S(n) else S(m);
  when 64 =>
    D(d) = if ConditionHolds(condition) then D(n) else D(m);
end;
```

C6.1.201 VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

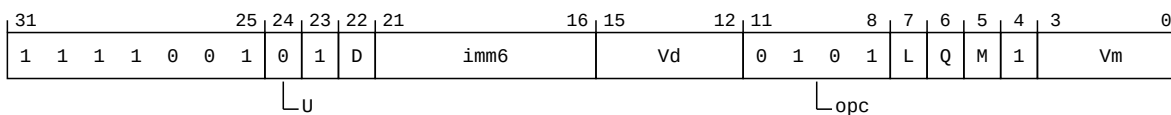
Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSHL{<c>}{<q>}.I<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSHL{<c>}{<q>}.I<size> {<Qd>, }<Qm>, #<imm>

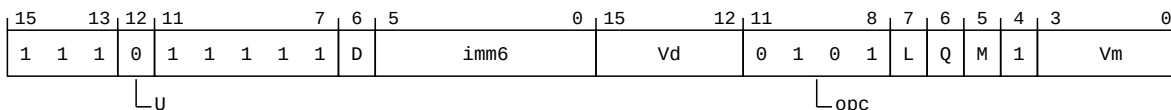
Decode for all variants of this encoding

```

if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSHL{<c>}{<q>}.I<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSHL{<c>}{<q>}.I<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 0 to <size>-1, encoded in the “imm6” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = LSL{esize} (D(m+r)[e*:esize], shift_amount);
        end;
    end;
end;
```

C6.1.202 VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

For a rounding shift, see [VRSHL](#).

The first operand and result elements are the same data type, and can be any one of:

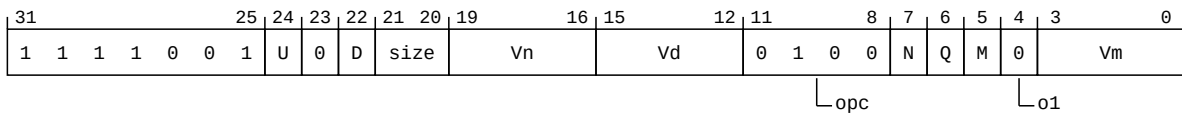
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

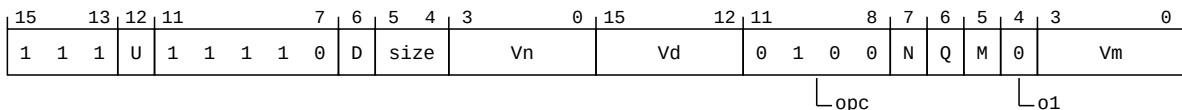
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSHL{<c>}{<q>}.<dt> {<Dd>, }<Dm>, <Dn>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSHL{<c>}{<q>}.<dt> {<Qd>, }<Qm>, <Qn>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1' || Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let n : integer = UInt(N::Vn);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shift : integer = SInt(D(n+r)[e*:esize][7:0]);
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            var result : integer;
            if shift >= 0 then
                result = element << shift;
            else
                result = element >> -shift;
            end;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;

```

C6.1.203 VSHLL

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:

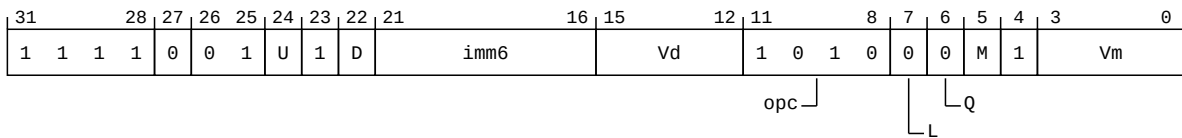
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit untyped integers, maximum shift only.

The result elements are twice the length of the operand elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding

Applies when (imm6 IN {100001, 10001x, 1001xx, 101xxx, 11xxxx, 010001, 01001x, 0101xx, 011xxx, 001001, 00101x, 0011xx})

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

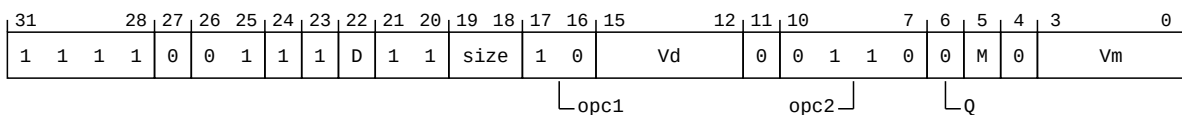
Decode for this encoding

```

if imm6 == '000xxx' then See("Related encodings"); end;
if imm6 IN {'001000', '010000', '100000'} then See("VMOVL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(imm6) - esize;
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);

```

A2



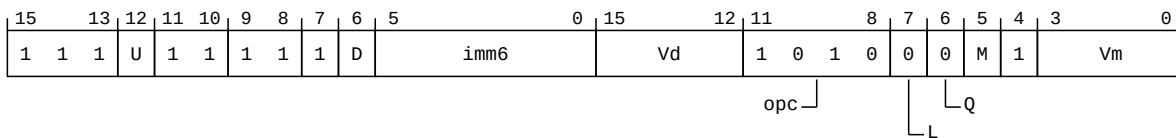
Encoding

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

Decode for this encoding

```
if size == '11' || Vd[0] == '1' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let shift_amount : integer = esize;
let unsigned : boolean = FALSE; // Or TRUE without change of functionality
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T1



Encoding

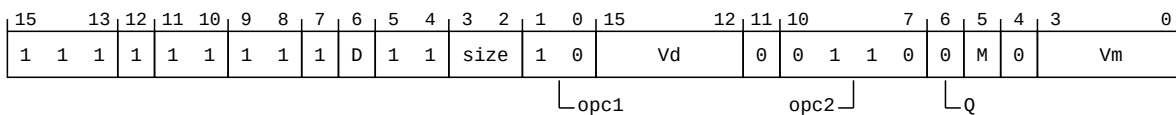
Applies when (imm6 IN {100001, 10001x, 1001xx, 101xxx, 11xxxx, 010001, 01001x, 0101xx, 011xxx, 001001, 00101x, 0011xx})

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

Decode for this encoding

```
if imm6 == '000xxx' then See("Related encodings"); end;
if imm6 IN {'001000', '010000', '100000'} then See("VMOVL"); end;
if Vd[0] == '1' then Undefined(); end;
let esize : integer() = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(imm6) - esize;
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

T2



Encoding

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

Decode for this encoding

```
if size == '11' || Vd[0] == '1' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let shift_amount : integer = esize;
let unsigned : boolean = FALSE; // Or TRUE without change of functionality
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” and “A2” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” and “T2” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

The data size for the elements of the operand. The following table shows the permitted values and their encodings:

<size>	Encoding T1/A1	Encoding T2/A2
8	Encoded as imm6<5:3> = 0b001	Encoded as size = 0b00
16	Encoded as imm6<5:4> = 0b01	Encoded as size = 0b01
32	Encoded as imm6<5> = 1	Encoded as size = 0b10

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

The immediate value. <imm> must lie in the range 1 to <size>, and:

- If <size> == <imm>, the encoding is T2/A2.
- Otherwise, the encoding is T1/A1, and:
 - If <size> == 8, <imm> is encoded in imm6<2:0>.
 - If <size> == 16, <imm> is encoded in imm6<3:0>.
 - If <size> == 32, <imm> is encoded in imm6<4:0>.

<type>

The data type for the elements of the operand. It must be one of:

- S** Signed. In encoding T1/A1, encoded as U = 0.
- U** Unsigned. In encoding T1/A1, encoded as U = 1.
- I** Untyped integer, Available only in encoding T2/A2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let opelt : bits(esize) = Din(m)[e*:esize];
        let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
        let result : integer = element << shift_amount;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

C6.1.204 VSHR

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see [VRSHR](#).

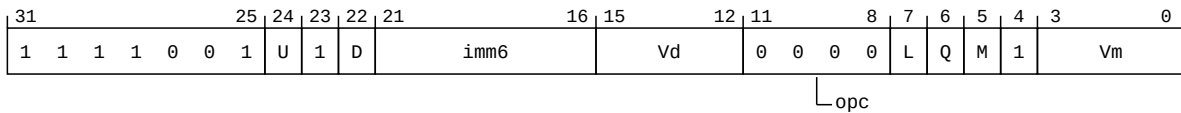
The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0 \ \&\& \ !(imm6[5:3] == 000 \ \&\& \ L == 0)$)

VSHR{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1 \ \&\& \ !(imm6[5:3] == 000 \ \&\& \ L == 0)$)

VSHR{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

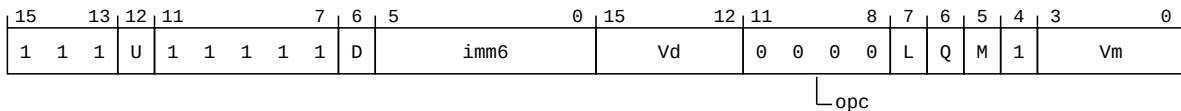
Decode for all variants of this encoding

```

if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0 \ \&\& \ !(imm6[5:3] == 000 \ \&\& \ L == 0)$)

VSHR{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSHR{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the “imm6” field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = element >> shift_amount;
            D(d+r)[e*:esize] = result[esize-1:0];
        end;
    end;
end;
```

C6.1.205 VSHR (zero)

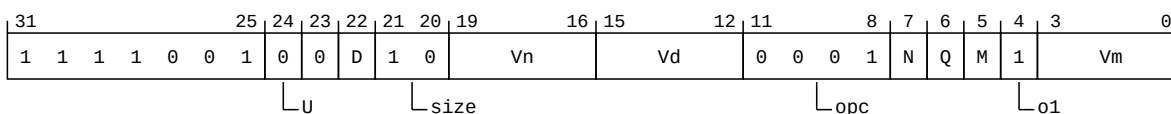
Vector Shift Right copies the contents of one SIMD register to another.

This is a pseudo-instruction of [VORR \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(register\)](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VORR \(register\)](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

Encoding for the 128-bit SIMD vector variant

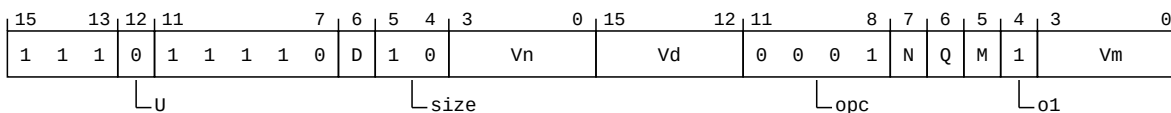
Applies when ($Q == 1$)

VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Dd>, }<Dm>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} {<Qd>, }<Qm>, <Qm>

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).
This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “N:Vn” and “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “N:Vn” and “M:Vm” field as <Qm>*2.

Operation

The description of [VORR \(register\)](#) gives the operational pseudocode for this instruction.

C6.1.206 VSHRN

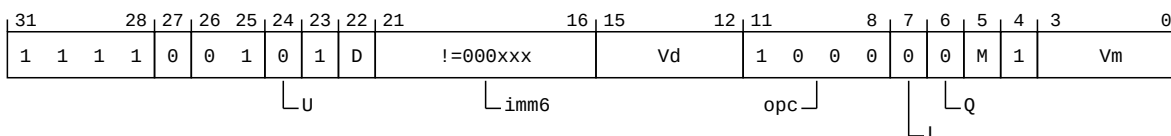
Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see [VRSHRN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

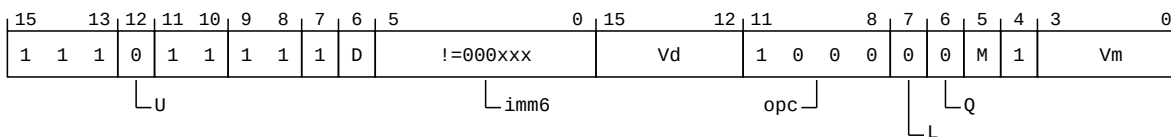
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

Decode for this encoding

```

if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
  
```

T1



Encoding

VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

Decode for this encoding

```

if imm6 == '000xxx' then See("Related encodings"); end;
if Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ(imm6[5:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (2 * esize) - UInt(imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
  
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size for the elements of the vectors, encoded in “imm6”:

imm6	<size>
001xxx	16
01xxxx	32
1xxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<imm>

Is an immediate value, in the range 1 to <size>/2, encoded in the “imm6” field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let result : bits(2*esize) = LSR(Qin(m>>1)[e*:(2*esize)], shift_amount);
        D(d)[e*esize] = result[esize-1:0];
    end;
end;
```

C6.1.207 VSHRN (zero)

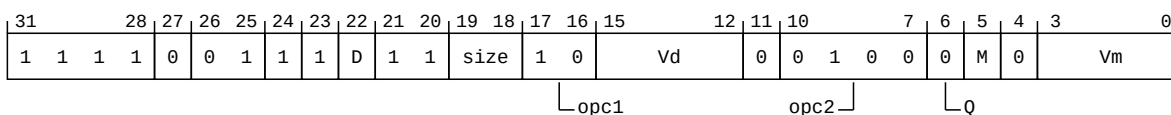
Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector.

This is a pseudo-instruction of **VMOVN**. This means:

- The encodings in this description are named to match the encodings of **VMOVN**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **VMOVN** gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (**A1**) and T32 (**T1**).

A1



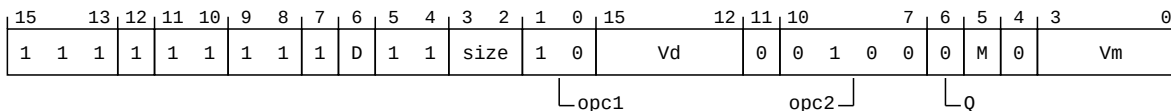
Encoding for the A1 variant

VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

T1



Encoding for the T1 variant

VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0

is equivalent to

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I 16
01	I 32
10	I 64
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

The description of [VMOVN](#) gives the operational pseudocode for this instruction.

C6.1.208 VSLI

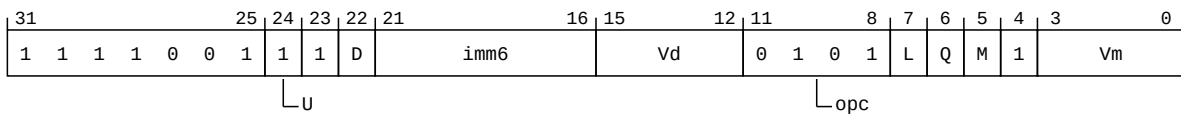
Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0 \ \&\& \ !(imm6[5:3] == 000 \ \&\& \ L == 0)$)

VSLI{<c>}{<q>}.<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1 \ \&\& \ !(imm6[5:3] == 000 \ \&\& \ L == 0)$)

VSLI{<c>}{<q>}.<size> {<Qd>, }<Qm>, #<imm>

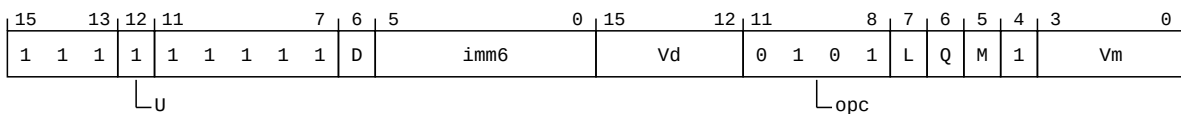
Decode for all variants of this encoding

```

if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0 \ \&\& \ !(imm6[5:3] == 000 \ \&\& \ L == 0)$)

VSLI{<c>}{<q>}.<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSLI{<c>}{<q>}.<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer() = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = UInt(L::imm6) - esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 0 to <size>-1, encoded in the “imm6” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let mask : bits(esize) = LSL{}(Ones(esize), shift_amount);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shifted_op : bits(esize) = LSL{}(D(m+r)[e*:esize], shift_amount);
            D(d+r)[e*:esize] = (D(d+r)[e*:esize] AND NOT(mask)) OR shifted_op;
        end;
    end;
end;
```

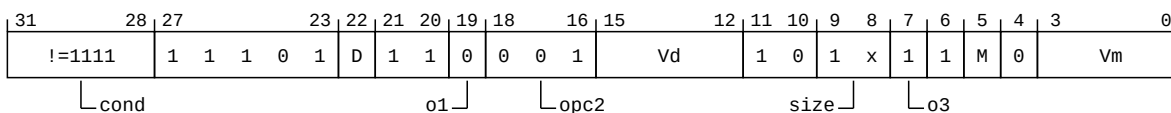
C6.1.209 VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

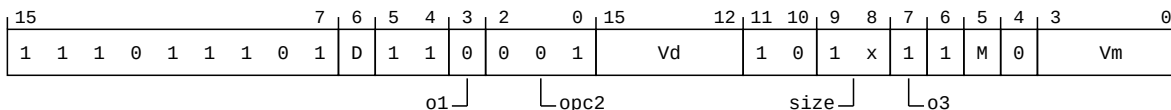
```
if size != '1x' then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>

Decode for all variants of this encoding

```
if size != '1x' then Undefined(); end;
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
let esize : integer() = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sm>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vm:M” field.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let fpcr : FPCR_Type = EffectiveFPCR();
    case esize of
        when 32 => S(d) = FPSqrt{32}(S(m), fpcr);
        when 64 => D(d) = FPSqrt{64}(D(m), fpcr);
    end;
end;
```

C6.1.210 VSRA

Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. For rounded results, see [VRSRA](#).

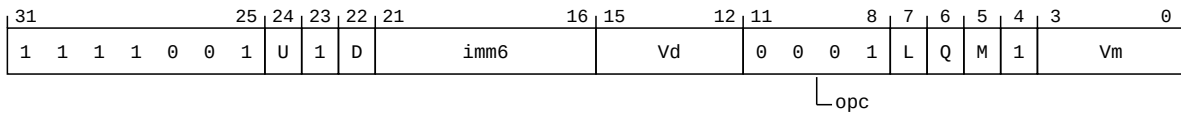
The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

```
VSRA{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

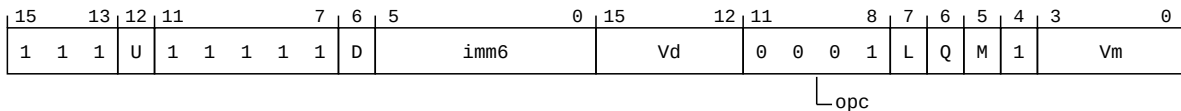
Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

```
VSRA{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

```
VSRA{<c>}{<q>}.<type><size> {<Dd>, }<Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSRA{<c>}{<q>}.<type><size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let unsigned : boolean = (U == '1');
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<type>

Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the “imm6” field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let opelt : bits(esize) = D(m+r)[e*:esize];
            let element : integer = if unsigned then UInt(opelt) else SInt(opelt);
            let result : integer = element >> shift_amount;
            D(d+r)[e*:esize] = D(d+r)[e*:esize] + result;
        end;
    end;
end;
end;

```

C6.1.211 VSRI

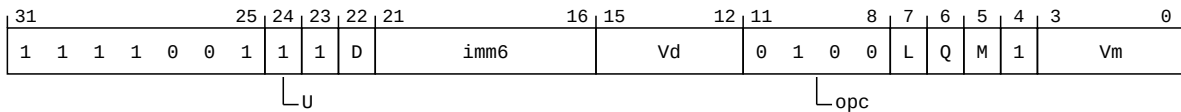
Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSRI{<c>}{<q>}.<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1 && !(imm6[5:3] == 000 && L == 0))

VSRI{<c>}{<q>}.<size> {<Qd>, }<Qm>, #<imm>

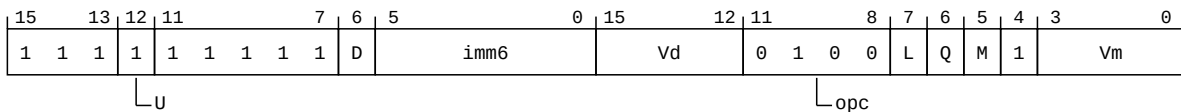
Decode for all variants of this encoding

```

if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0 && !(imm6[5:3] == 000 && L == 0))

VSRI{<c>}{<q>}.<size> {<Dd>, }<Dm>, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when $(Q == 1 \ \&\& \ !(\text{imm6}[5:3] == 000 \ \&\& \ L == 0))$

VSRI{<c>}{<q>}.<size> {<Qd>, }<Qm>, #<imm>

Decode for all variants of this encoding

```
if (L::imm6) == '0000xxx' then See("Related encodings"); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer() = 8 << HighestSetBitNZ((L::imm6)[6:3]);
let elements : integer = 64 DIV esize;
let shift_amount : integer = (esize * 2) - UInt(L::imm6);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Related encodings: See [Advanced SIMD one register and modified immediate](#) for the T32 instruction set, or [Advanced SIMD one register and modified immediate](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size for the elements of the vectors, encoded in “L:imm6”:

L	imm6	<size>
0	001xxx	8
0	01xxxx	16
0	1xxxxxx	32
1	xxxxxxx	64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<imm>

Is an immediate value, in the range 1 to <size>, encoded in the “imm6” field as <size> - <imm>.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let mask : bits(esize) = LSR(Ones{esize}, shift_amount);
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            let shifted_op : bits(esize) = LSR(D(m+r)[e*:esize], shift_amount);
            D(d+r)[e*:esize] = (D(d+r)[e*:esize] AND NOT(mask)) OR shifted_op;
        end;
    end;
end;
```

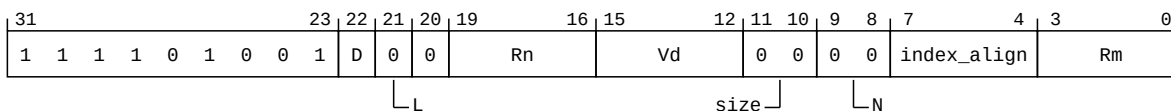
C6.1.212 VST1 (single element from one lane)

Store single element from one lane of one register stores one element to memory from one element of a register. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VST1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

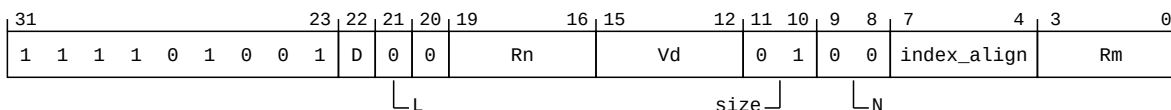
Decode for all variants of this encoding

```

if size == '1l' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

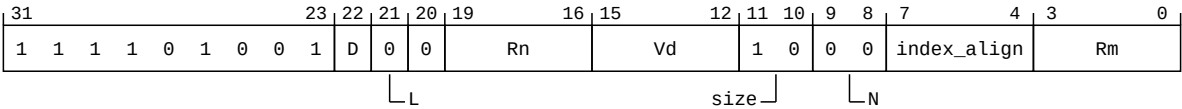
Applies when (Rm != 11x1)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

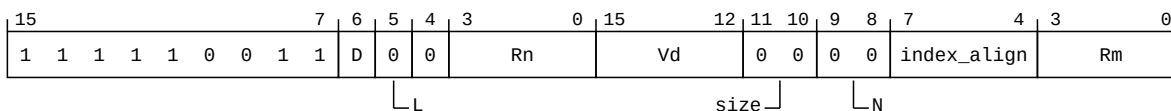
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D:Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VST1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

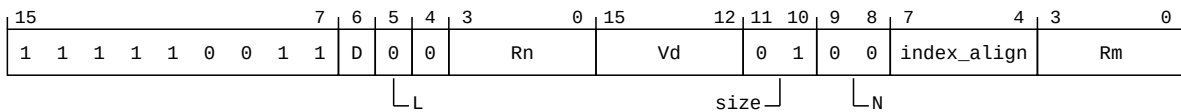
if size == '1l' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let alignment : integer{} = 1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);

let wback : boolean = (m != 15);

let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [(<Rn>{:<align>})], <Rm>

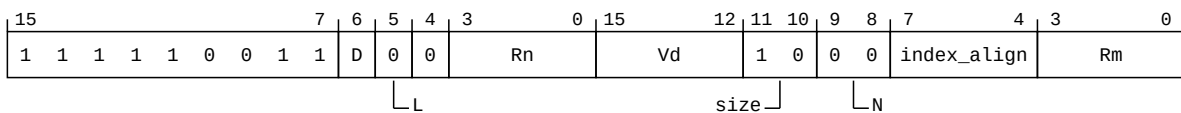
Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[2] != '0' then Undefined(); end;
if index_align[1:0] != '00' && index_align[1:0] != '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the single 64-bit name of the SIMD&FP register holding the element.

The list must be { <Dd>[<index>] }.

The register <Dd> is encoded in the “D:Vd” field.

The permitted values and encoding of <index> depend on <size>:

<size> == **8** <index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.

<size> == **16** <index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.

<size> == **32** <index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

When <size> == 8, <align> must be omitted, otherwise it is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> == 8 Encoded in the “index_align<0>” field as 0.
<size> == 16 Encoded in the “index_align<1:0>” field as 0b00.
<size> == 32 Encoded in the “index_align<2:0>” field as 0b000.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16 <align> is 16, meaning 16-bit alignment, encoded in the “index_align<1:0>” field as 0b01.
<size> == 32 <align> is 32, meaning 32-bit alignment, encoded in the “index_align<2:0>” field as 0b011.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    MemU{8*bytes}(address) = D(d)[index*:(8*bytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + ebytes;
        end;
    end;
end;
end;

```

C6.1.213 VST1 (multiple single elements)

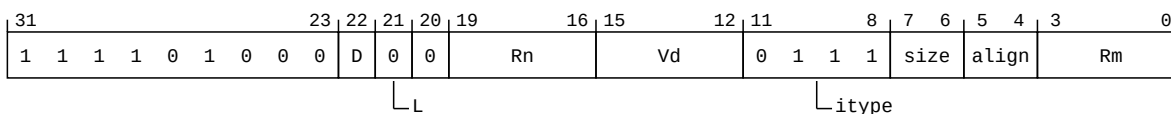
Store multiple single elements from one, two, three, or four registers

Store multiple single elements from one, two, three, or four registers stores elements to memory from one, two, three, or four registers, without interleaving. Every element of each register is stored. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1, A2, A3, and A4) and T32 (T1, T2, T3, and T4).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

$$\text{VST1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{\langle \text{align} \rangle\}]!$$

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

$$\text{VST1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

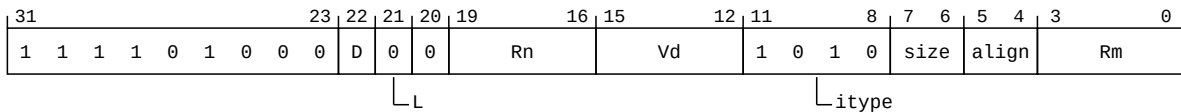
Decode for all variants of this encoding

```

if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```


A2



Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [(<Rn>{:<align>})], <Rm>

Decode for all variants of this encoding

```

if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(aligned);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

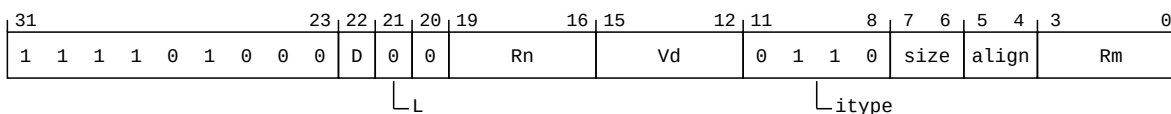
```

CONSTRAINED UNPREDICTABLE behavior

If $\text{d+regs} > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

$$\text{VST1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

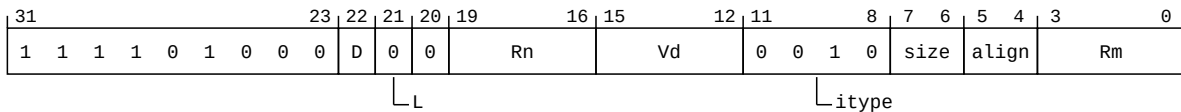
```

CONSTRAINED UNPREDICTABLE behavior

If $d + \text{regs} > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

A4



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

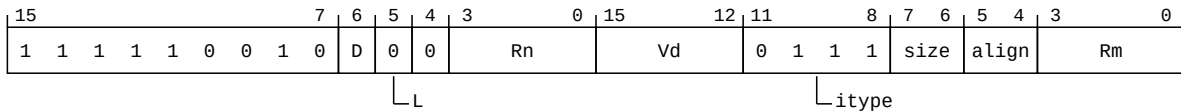
```
let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [(<Rn>{:<align>})], <Rm>

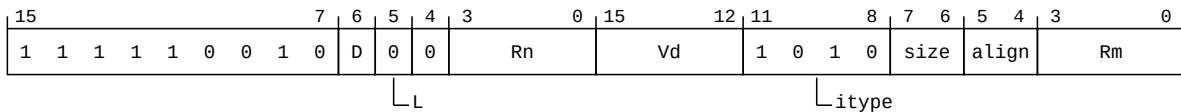
Decode for all variants of this encoding

```

if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 1;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 then UnpredictableProcedure(); end;

```

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

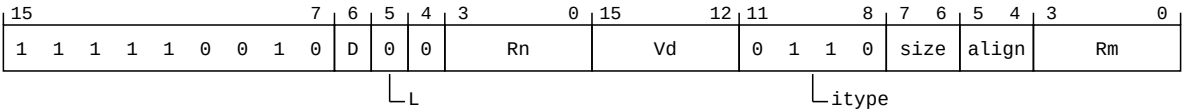
```
if align == '11' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 2;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d+regs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if align[1] == '1' then Undefined(); end;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let regs : integer = 3;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

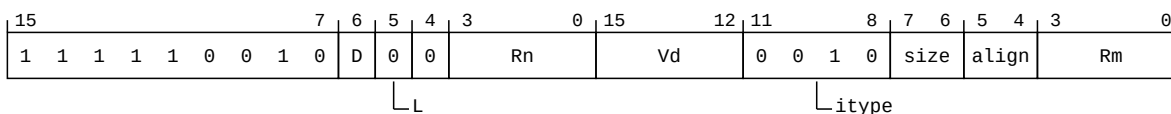
```

CONSTRAINED UNPREDICTABLE behavior

If $d + \text{regs} > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

T4



Encoding for the Offset variant

Applies when (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

$$\text{VST1}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$
Decode for all variants of this encoding

```

let regs : integer = 4;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 & m != 13);
if n == 15 || d+regs > 32 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST1 \(multiple single elements\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

`<c>`

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, “A3 Post-indexed”, “A4 Offset”, and “A4 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, “T3 Post-indexed”, “T4 Offset”, and “T4 Post-indexed” variants: see [Standard assembler syntax fields](#).

`<q>`

See [Standard assembler syntax fields](#).

`<size>`

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	64

`<list>`

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- { `<Dd>` } Single register. Selects the A1 and T1 encodings of the instruction.
- { `<Dd>`, `<Dd+1>` } Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.
- { `<Dd>`, `<Dd+1>`, `<Dd+2>` } Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.
- { `<Dd>`, `<Dd+1>`, `<Dd+2>`, `<Dd+3>` } Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register `<Dd>` is encoded in the “D:Vd” field.

`<Rn>`

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “align” field as 0b00.

Whenever <align> is present, the permitted values are:

- 64** 64-bit alignment, encoded in the “align” field as 0b01.
- 128** 128-bit alignment, encoded in the “align” field as 0b10. Available only if <list> contains two or four registers.
- 256** 256-bit alignment, encoded in the “align” field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about <Rn>, !, and <Rm>, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if ebytes != 8 then
                MemU{8*ebytes}(address) = D(d+r)[e*(8*ebytes)];
            else
                if !IsAlignedSize(address, ebytes) && AlignmentEnforced() then
                    let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
                    AArch32_Abort(fault);
                end;

                let data : bits(64) = D(d+r)[e*64];
                if BigEndian(AccessType_ASIMD) then
                    MemU{32}(address) = data[63:32];
                    MemU{32}(address+4) = data[31:0];
                else
                    MemU{32}(address) = data[31:0];
                    MemU{32}(address+4) = data[63:32];
                end;
            end;
            address = address + ebytes;
        end;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 8*regs;
        end;
    end;
```



```
        end;  
    end;  
end;
```

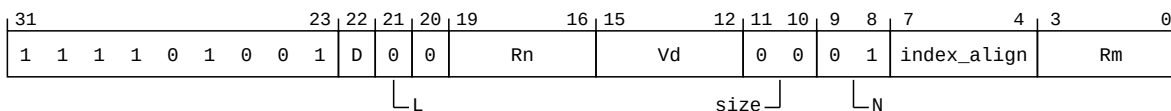
C6.1.214 VST2 (single 2-element structure from one lane)

Store single 2-element structure from one lane of two registers stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

$$\text{VST2}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{\langle \text{align} \rangle\}]!$$

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VST2}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;

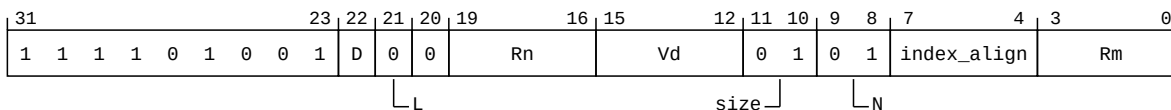
```

CONSTRAINED UNPREDICTABLE behavior

If $d_2 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

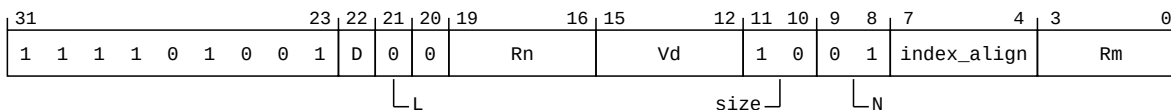
```
if size == 'l1' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

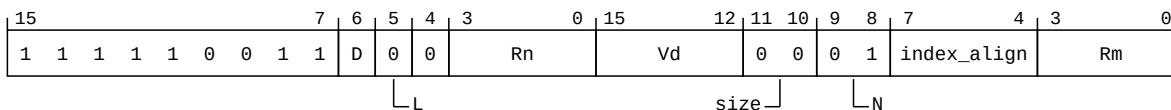
```
if size == 'l1' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

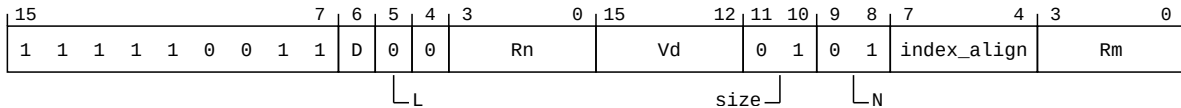
```
if size == 'l1' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

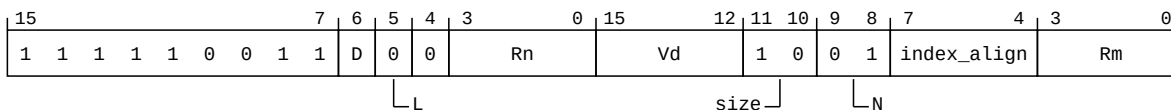
```
if size == 'l1' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == 'l1' then Undefined(); end;
if index_align[1] != '0' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST2 \(single 2-element structure from one lane\)](#).

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>] } Single-spaced registers, encoded as “spacing” = 0.
{ <Dd>[<index>], <Dd+2>[<index>] } Double-spaced registers, encoded as “spacing” = 1. Not permitted when <size> = 8.

The encoding of “spacing” depends on <size>:

<size> = 16 “spacing” is encoded in the “index_align<1>” field.
<size> = 32 “spacing” is encoded in the “index_align<2>” field.

The register <Dd> is encoded in the “D:Vd” field.

The permitted values and encoding of <index> depend on <size>:

<size> = 8 <index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.
<size> = 16 <index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.
<size> = 32 <index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> = 8 Encoded in the “index_align<0>” field as 0.
<size> = 16 Encoded in the “index_align<0>” field as 0.
<size> = 32 Encoded in the “index_align<1:0>” field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

$\langle \text{size} \rangle == 8$ $\langle \text{align} \rangle$ is 16, meaning 16-bit alignment, encoded in the “index_align<0>” field as 1.
 $\langle \text{size} \rangle == 16$ $\langle \text{align} \rangle$ is 32, meaning 32-bit alignment, encoded in the “index_align<0>” field as 1.
 $\langle \text{size} \rangle == 32$ $\langle \text{align} \rangle$ is 64, meaning 64-bit alignment, encoded in the “index_align<1:0>” field as 0b01.

: is the preferred separator before the $\langle \text{align} \rangle$ value, but the alignment can be specified as @ $\langle \text{align} \rangle$, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    MemU{8*bytes}(address) = D(d)[index*:(8*bytes)];
    MemU{8*bytes}(address+bytes) = D(d2)[index*:(8*bytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 2*bytes;
        end;
    end;
end;
end;

```

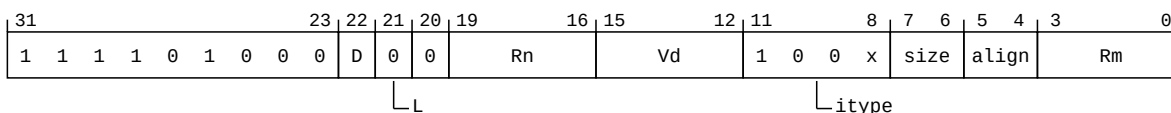
C6.1.215 VST2 (multiple 2-element structures)

Store multiple 2-element structures from two or four registers stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see [Element and structure load/store instructions](#). Every element of each register is saved. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

$$\text{VST2}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;

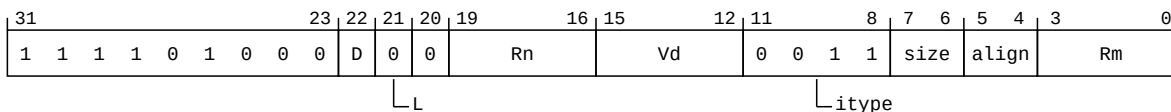
```

CONSTRAINED UNPREDICTABLE behavior

If $d_{2+\text{pairs}} > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

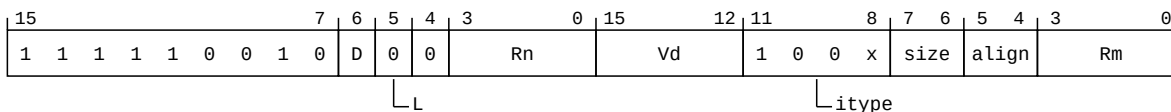
```
if size == 'l1' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

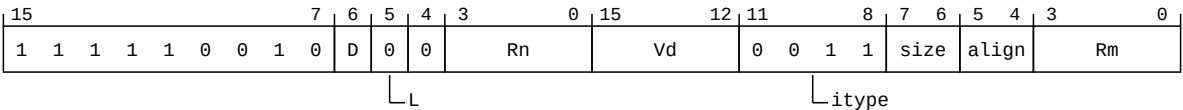
```
if align == '11' then Undefined(); end;
if size == '11' then Undefined(); end;
let pairs : integer = 1;
let inc : integer = if itype == '1001' then 2 else 1;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
let pairs : integer = 2;
let inc : integer = 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d2+pairs > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST2 \(multiple 2-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, and “A2 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, and “T2 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- { <Dd>, <Dd+1> } Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as 0b1000.
- { <Dd>, <Dd+2> } Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the “itype” field as 0b1001.
- { <Dd>, <Dd+1>, <Dd+2>, <Dd+3> } Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “align” field as 0b00.

Whenever <align> is present, the permitted values are:

- 64** 64-bit alignment, encoded in the “align” field as 0b01.
- 128** 128-bit alignment, encoded in the “align” field as 0b10.
- 256** 256-bit alignment, encoded in the “align” field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for r = 0 to pairs-1 do
        for e = 0 to elements-1 do
            MemU{8*bytes}(address      ) = D(d+r)[e:(8*bytes)];
            MemU{8*bytes}(address+ebytes) = D(d2+r)[e:(8*bytes)];
            address = address + 2*bytes;
        end;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 16*pairs;
        end;
    end;
end;

```

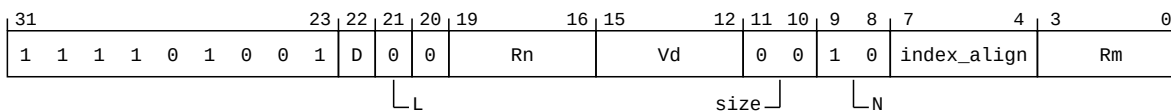
C6.1.216 VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1, A2, and A3) and T32 (T1, T2, and T3).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>} {<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

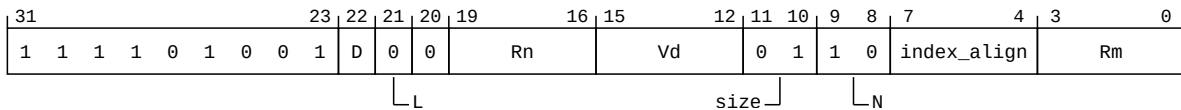
```

CONSTRAINED UNPREDICTABLE behavior

If $d_3 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

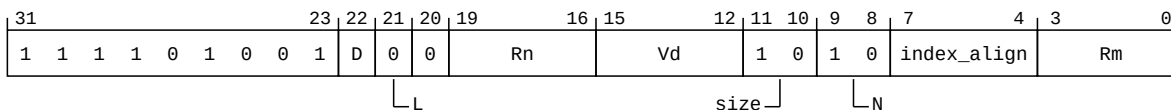
```

CONSTRAINED UNPREDICTABLE behavior

If $d_3 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

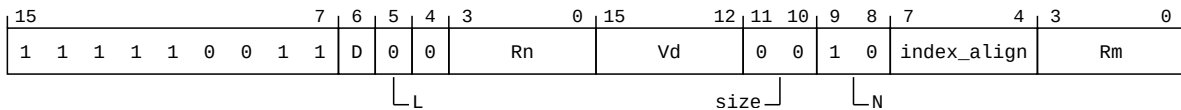
```
if size == 'l1' then Undefined(); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

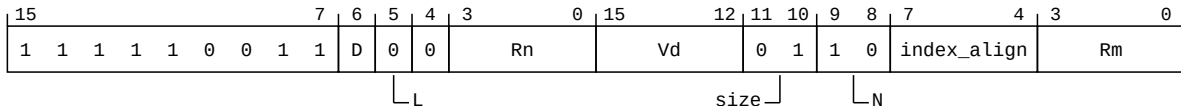
```

CONSTRAINED UNPREDICTABLE behavior

If $d_3 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

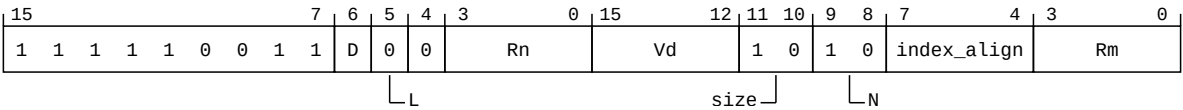
```
if size == 'l1' then Undefined(); end;
if index_align[0] != '0' then Undefined(); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if index_align[1:0] != '00' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d3 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST3 \(single 3-element structure from one lane\)](#).

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.

The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] } Single-spaced registers, encoded as “spacing” = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] } Double-spaced registers, encoded as “spacing” = 1. Not permitted when <size> == 8.

The encoding of “spacing” depends on <size>:

<size> == 8 “spacing” is encoded in the “index_align<0>” field.

<size> == 16 “spacing” is encoded in the “index_align<1>” field, and “index_align<0>” is set to 0.

<size> == 32 “spacing” is encoded in the “index_align<2>” field, and “index_align<1:0>” is set to 0b00.

The register <Dd> is encoded in the “D:Vd” field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8 <index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.

<size> == 16 <index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.

<size> == 32 <index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Alignment

Standard alignment rules apply, see Alignment support.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let address : bits(32) = R(n);
    MemU{8*bytes}(address) = D(d)[index*:(8*bytes)];
    MemU{8*bytes}(address+ebytes) = D(d2)[index*:(8*bytes)];
    MemU{8*bytes}(address+2*ebytes) = D(d3)[index*:(8*bytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 3*ebytes;
        end;
    end;
end;
```

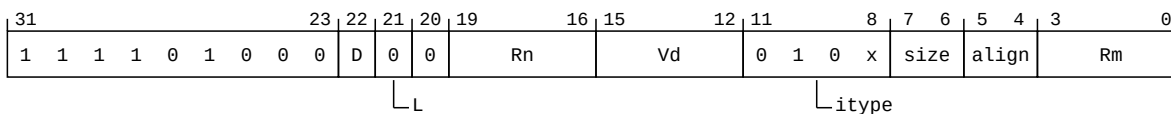
C6.1.217 VST3 (multiple 3-element structures)

Store multiple 3-element structures from three registers stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see [Element and structure load/store instructions](#). Every element of each register is saved. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

$$\text{VST3}\{\langle c \rangle\}\{\langle q \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if size == '11' || align[1] == '1' then Undefined(); end;
var inc : integer;
case itype of
  when '0100' =>
    inc = 1;
  when '0101' =>
    inc = 2;
  otherwise =>
    See("Related encodings");
end;

let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

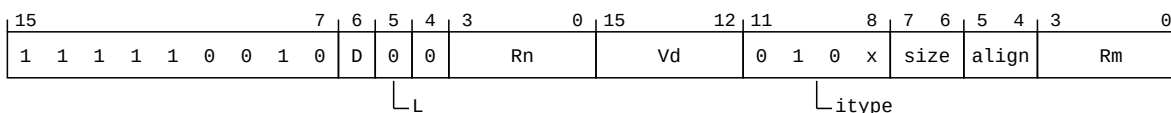
```


CONSTRAINED UNPREDICTABLE behavior

If $d3 > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Encoding for the Offset variant

Applies when $(Rm == 1111)$

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when $(Rm == 1101)$

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when $(Rm != 11x1)$

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' || align[1] == '1' then Undefined(); end;
var inc : integer;
case itype of
  when '0100' =>
    inc = 1;
  when '0101' =>
    inc = 2;
  otherwise =>
    See("Related encodings");
end;
let alignment : integer{} = if align[0] == '0' then 1 else 8;
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $d3 > 31$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The memory locations specified by the instruction and the number of registers specified by the instruction become **UNKNOWN**. If the instruction specifies writeback, then that register becomes **UNKNOWN**. This behavior does not affect any other memory locations.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST3 \(multiple 3-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1 Offset” and “A1 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- { <Dd>, <Dd+1>, <Dd+2> } Single-spaced registers, encoded in the “itype” field as 0b0100.
- { <Dd>, <Dd+2>, <Dd+4> } Double-spaced registers, encoded in the “itype” field as 0b0101.

The register <Dd> is encoded in the “D:Vd” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “align” field as 0b00.

Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the “align” field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for e = 0 to elements-1 do
        MemU{8*bytes}(address) = D(d)[e:(8*bytes)];
        MemU{8*bytes}(address+ebytes) = D(d2)[e:(8*bytes)];
        MemU{8*bytes}(address+2*ebytes) = D(d3)[e:(8*bytes)];
        address = address + 3*ebytes;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 24;
        end;
    end;
end;
```

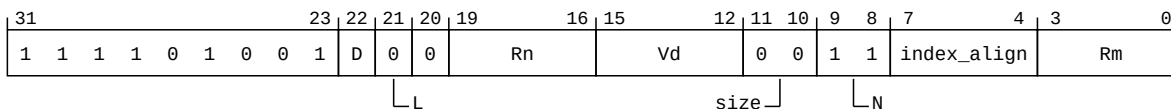
C6.1.218 VST4 (single 4-element structure from one lane)

Store single 4-element structure from one lane of four registers stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#), [A2](#), and [A3](#)) and T32 ([T1](#), [T2](#), and [T3](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if size != '00' then See("Related encodings"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

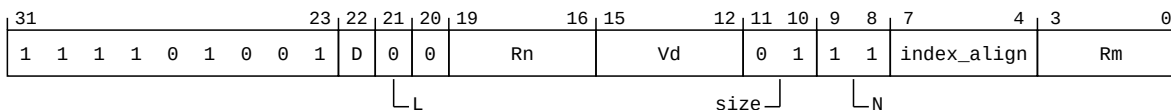
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

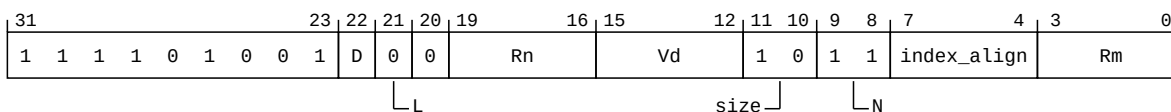
```
if size == 'l1' then Undefined(); end;
if size != '01' then See("Related encodings"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3



Encoding for the Offset variant

Applies when (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if size != '10' then See("Related encodings"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

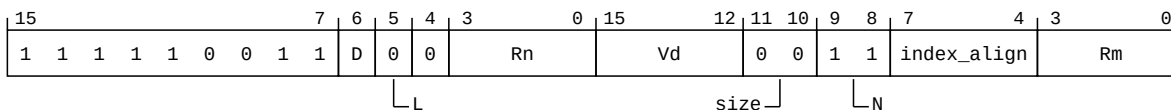
```

CONSTRAINED UNPREDICTABLE behavior

If $d_4 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

T1



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

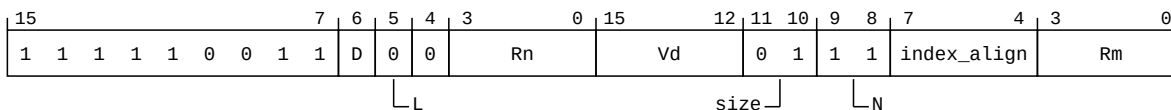
```
if size == 'l1' then Undefined(); end;
if size != '00' then See("Related encodings"); end;
let ebytes : integer{} = 1;
let index : integer = UInt(index_align[3:1]);
let inc : integer = 1;
let alignment : integer{} = if index_align[0] == '0' then 1 else 4;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Encoding for the Offset variant

Applies when (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

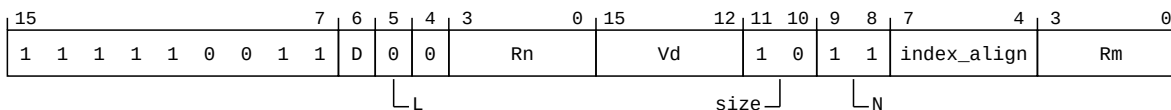
```
if size == 'l1' then Undefined(); end;
if size != '01' then See("Related encodings"); end;
let ebytes : integer{} = 2;
let index : integer = UInt(index_align[3:2]);
let inc : integer = if index_align[1] == '0' then 1 else 2;
let alignment : integer{} = if index_align[0] == '0' then 1 else 8;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If d4 > 31, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Encoding for the Offset variant

Applies when (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

$$\text{VST4}\{\langle c \rangle\}\{\langle g \rangle\}.\langle \text{size} \rangle \quad \langle \text{list} \rangle, \quad [\langle \text{Rn} \rangle\{:\langle \text{align} \rangle\}], \quad \langle \text{Rm} \rangle$$

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if size != '10' then See("Related encodings"); end;
if index_align[1:0] == '11' then Undefined(); end;
let ebytes : integer{} = 4;
let index : integer = UInt(index_align[3]);
let inc : integer = if index_align[2] == '0' then 1 else 2;
let alignment : integer{} = if index_align[1:0] == '00' then 1 else 4 << UInt(index_align[1:0]);
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If $d_4 > 31$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST4 \(single 4-element structure from one lane\)](#).

Assembler Symbols

<c>

For the “A1 Offset”, “A1 Post-indexed”, “A2 Offset”, “A2 Post-indexed”, “A3 Offset”, and “A3 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset”, “T1 Post-indexed”, “T2 Offset”, “T2 Post-indexed”, “T3 Offset”, and “T3 Post-indexed” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list>

Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.

The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] } Single-spaced registers, encoded as “spacing” = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] } Double-spaced registers, encoded as “spacing” = 1. Not permitted when <size> == 8.

The encoding of “spacing” depends on <size>:

<size> == 16 “spacing” is encoded in the “index_align<1>” field.

<size> == 32 “spacing” is encoded in the “index_align<2>” field.

The register <Dd> is encoded in the “D:Vd” field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8 <index> is in the range 0 to 7, encoded in the “index_align<3:1>” field.

<size> == 16 <index> is in the range 0 to 3, encoded in the “index_align<3:2>” field.

<size> == 32 <index> is 0 or 1, encoded in the “index_align<3>” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field.

<align>

Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and the encoding depends on <size>:

<size> == 8 Encoded in the “index_align<0>” field as 0.

<size> == 16 Encoded in the “index_align<0>” field as 0.

<size> == 32 Encoded in the “index_align<1:0>” field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == **8** <align> is 32, meaning 32-bit alignment, encoded in the “index_align<0>” field as 1.
<size> == **16** <align> is 64, meaning 64-bit alignment, encoded in the “index_align<0>” field as 1.
<size> == **32** <align> can be 64 or 128. 64-bit alignment is encoded in the “index_align<1:0>” field as 0b01,
and 128-bit alignment is encoded in the “index_align<1:0>” field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    let address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    MemU{8*bytes}(address) = D(d)[index*:(8*bytes)];
    MemU{8*bytes}(address+bytes) = D(d2)[index*:(8*bytes)];
    MemU{8*bytes}(address+2*bytes) = D(d3)[index*:(8*bytes)];
    MemU{8*bytes}(address+3*bytes) = D(d4)[index*:(8*bytes)];
    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 4*bytes;
        end;
    end;
end;
end;

```

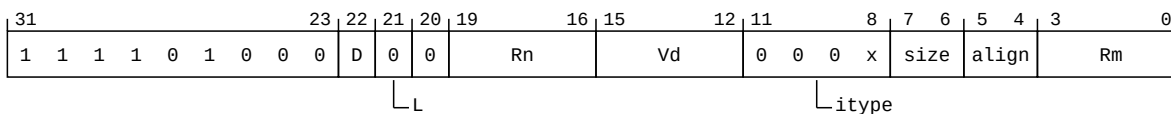
C6.1.219 VST4 (multiple 4-element structures)

Store multiple 4-element structures from four registers stores multiple 4-element structures to memory from four registers, with interleaving. For more information, see [Element and structure load/store instructions](#). Every element of each register is saved. For details of the addressing mode, see [Advanced SIMD addressing mode](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Offset variant

Applies when (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when ($R_m \neq 11 \times 1$)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Decode for all variants of this encoding

```

if size == '11' then Undefined(); end;
if itype != '000x' then See("Related encodings"); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D:Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;

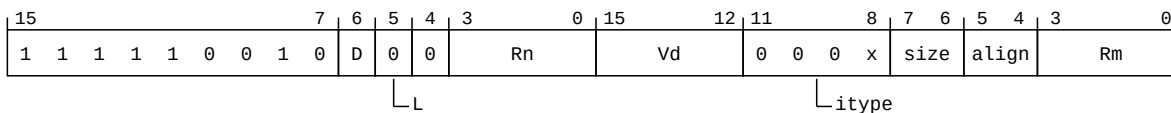
```

CONSTRAINED UNPREDICTABLE behavior

If $d4 > 31$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Encoding for the Offset variant

Applies when $(Rm == 1111)$

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when $(Rm == 1101)$

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Encoding for the Post-indexed variant

Applies when $(Rm != 11x1)$

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if itype != '000x' then See("Related encodings"); end;
let inc : integer = if itype[0] == '0' then 1 else 2;
let alignment : integer{} = if align == '00' then 1 else 4 << UInt(align);
let ebytes : integer{} = 1 << UInt(size);
let elements : integer = 8 DIV ebytes;
let d : integer = UInt(D::Vd);
let d2 : integer = d + inc;
let d3 : integer = d2 + inc;
let d4 : integer = d3 + inc;
let n : integer = UInt(Rn);
let m : integer = UInt(Rm);
let wback : boolean = (m != 15);
let register_index : boolean = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

For more information about the `CONSTRAINED UNPREDICTABLE` behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST4 \(multiple 4-element structures\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

`<c>`

For the “A1 Offset” and “A1 Post-indexed” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 Offset” and “T1 Post-indexed” variants: see [Standard assembler syntax fields](#).

`<q>`

See [Standard assembler syntax fields](#).

`<size>`

Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

`<list>`

Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

- { `<Dd>`, `<Dd+1>`, `<Dd+2>`, `<Dd+3>` } Single-spaced registers, encoded in the “itype” field as `0b0000`.
- { `<Dd>`, `<Dd+2>`, `<Dd+4>`, `<Dd+6>` } Double-spaced registers, encoded in the “itype” field as `0b0001`.

The register `<Dd>` is encoded in the “D:Vd” field.

`<Rn>`

Is the general-purpose base register, encoded in the “Rn” field.

`<align>`

Is the optional alignment.

Whenever `<align>` is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the “align” field as `0b00`.

Whenever `<align>` is present, the permitted values are:

- 64** 64-bit alignment, encoded in the “align” field as 0b01.
- 128** 128-bit alignment, encoded in the “align” field as 0b10.
- 256** 256-bit alignment, encoded in the “align” field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm>

Is the general-purpose index register containing an offset applied after the access, encoded in the “Rm” field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    var address : bits(32) = R(n);

    let nontemporal : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
    if !IsAlignedSize(address, alignment) then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    for e = 0 to elements-1 do
        MemU{8*abytes}(address) = D(d)[e:(8*abytes)];
        MemU{8*abytes}(address+abytes) = D(d2)[e:(8*abytes)];
        MemU{8*abytes}(address+2*abytes) = D(d3)[e:(8*abytes)];
        MemU{8*abytes}(address+3*abytes) = D(d4)[e:(8*abytes)];
        address = address + 4*abytes;
    end;

    if wback then
        if register_index then
            R(n) = R(n) + R(m);
        else
            R(n) = R(n) + 32;
        end;
    end;
end;

```

C6.1.220 VSTM, VSTMDB, VSTMIA

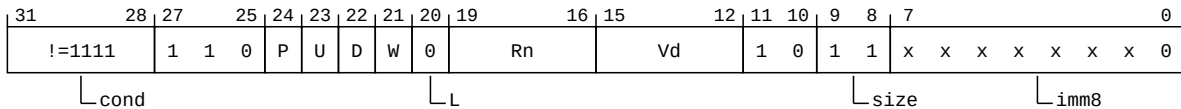
Store multiple SIMD&FP registers stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register.

Depending on settings in the CPACR, NSACR, HCPTR, and FPExc registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the alias VPUSH.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2).

A1



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist> // (Preferred syntax)

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{(imm8::'00')};
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

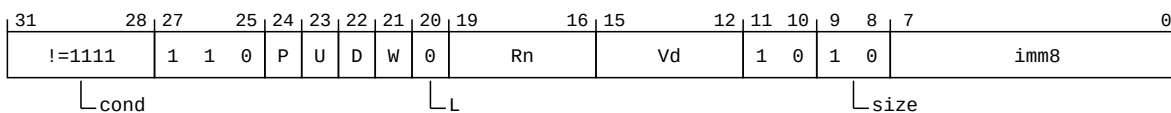
If regs == 0, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If regs > 16 || (d+regs) > 32, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

A2



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> // (Preferred syntax)
```

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

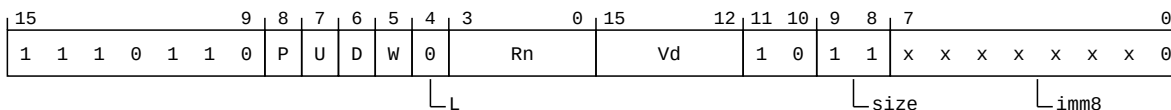
If $\text{reqs} == 0$, then one of the following behaviors must occur:

- The instruction is **UNDEFINED**.
- The instruction executes as **NOP**.
- The instruction operates as a **VSTM** with the same addressing mode but stores no registers.

If $(d + \text{reqs}) > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

T1



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist> // (Preferred syntax)
```

VSTMIA{<c>}{<q>}{,<size>} <Rn>{!}, <dreglist>

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;

// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = FALSE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(D::Vd);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8) DIVRM 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || regs > 16 || (d+regs) > 32 then UnpredictableProcedure(); end;
if imm8[0] == '1' && (d+regs) > 16 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

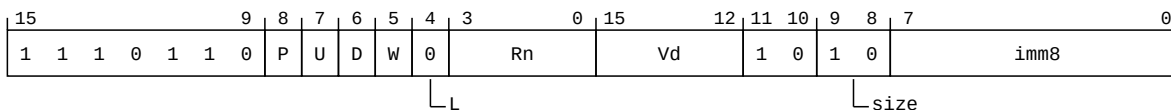
If `regs == 0`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

If $\text{regs} > 16 \ || \ (\text{d+regs}) > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

T2



Encoding for the Decrement Before variant

Applies when (P == 1 && U == 0 && W == 1)

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>

Encoding for the Increment After variant

Applies when (P == 0 && U == 1)

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist> // (Preferred syntax)

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

Decode for all variants of this encoding

```

if P == '0' && U == '0' && W == '0' then See("Related encodings"); end;
if P == '1' && W == '0' then See("VSTR"); end;
if P == U && W == '1' then Undefined(); end;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
let single_regs : boolean = TRUE;
let add : boolean = (U == '1');
let wback : boolean = (W == '1');
let d : integer = UInt(Vd::D);
let n : integer = UInt(Rn);
let imm32 : bits(32) = ZeroExtend{}(imm8::'00');
let regs : integer = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UnpredictableProcedure(); end;
if regs == 0 || (d+regs) > 32 then UnpredictableProcedure(); end;

```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

If $(d + \text{regs}) > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become `UNKNOWN`. If the instruction specifies writeback, then that register becomes `UNKNOWN`. This behavior does not affect any other memory locations.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VSTM](#).

Related encodings: See [Advanced SIMD and floating-point 64-bit move](#) for the T32 instruction set, or [Advanced SIMD and floating-point 64-bit move](#) for the A32 instruction set.

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<size>

An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

<dreglist>

Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “D:Vd”, and “imm8” is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

!

Specifies base register writeback, encoded in “W”:

W	!
0	[absent]
1	[present]

<sreglist>

Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in “Vd:D”, and “imm8” is set to the number of registers in the list. The list must contain at least one register.

Alias Conditions

Alias	Is preferred when
VPUSH	P == '1' && U == '0' && W == '1' && Rn == '1101'

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    var address : bits(32) = if add then R(n) else R(n)-imm32;
    for r = 0 to regs-1 do
        if single_regs then
            MemA{32}(address) = S(d+r);
            address = address+4;
        else
```

```
        // Store as two word-aligned words in the correct order for current endianness.
        if BigEndian(AccessType_ASIMD) then
            MemA{32}(address) = D(d+r)[63:32];
            MemA{32}(address+4) = D(d+r)[31:0];
        else
            MemA{32}(address) = D(d+r)[31:0];
            MemA{32}(address+4) = D(d+r)[63:32];
        end;

        address = address+8;
    end;
end;

if wback then R(n) = if add then R(n)+imm32 else R(n)-imm32; end;
end;
```

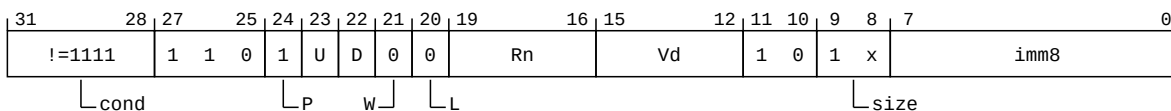
C6.1.221 VSTR

Store SIMD&FP register stores a single register from the Advanced SIMD and floating-point register file to memory, using an address from a general-purpose register, with an optional offset.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #<+/-><imm>}]
```

Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #<+/-><imm>}]
```

Decode for all variants of this encoding

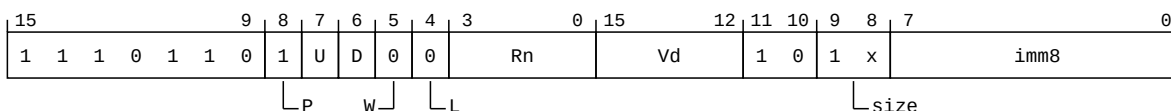
```
if size IN { '00', '01' } then Undefined(); end;
let add : boolean = (U == '1');
let esize : integer = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #<+/-><imm>}]

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #<+/-><imm>}]

Decode for all variants of this encoding

```
if size IN { '00', '01' } then Undefined(); end;
let add : boolean = (U == '1');
let esize : integer{} = 8 << UInt(size);
let imm32 : integer = UInt(imm8) << (if size == '01' then 1 else 2);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>

See [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

.32

Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Sd>

Is the 32-bit name of the SIMD&FP source register, encoded in the “Vd:D” field.

<Rn>

Is the general-purpose base register, encoded in the “Rn” field. The PC can be used, but this is deprecated.

+/-

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>

is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the “imm8” field as <imm>/4.

.64

Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.

<Dd>

Is the 64-bit name of the SIMD&FP source register, encoded in the “D:Vd” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    let address : bits(32) = if add then (R(n) + imm32) else (R(n) - imm32);
    case esize of
        when 32 =>
            MemA{32}(address) = S(d);
        when 64 =>
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA{32}(address) = D(d)[63:32];
                MemA{32}(address+4) = D(d)[31:0];
            else
                MemA{32}(address) = D(d)[31:0];
                MemA{32}(address+4) = D(d)[63:32];
            end;
    end;
end;
```

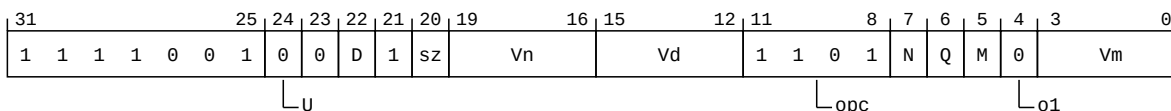

C6.1.222 VSUB (floating-point)

Vector Subtract (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

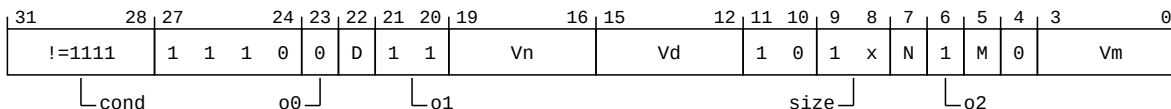
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

A2



Encoding for the Single-precision scalar variant

Applies when (size == 10)

VSUB{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>

Encoding for the Double-precision scalar variant

Applies when (size == 11)

VSUB{<c>} {<q>}.F64 {<Dd>, }<Dn>, <Dm>

Decode for all variants of this encoding

```

if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;

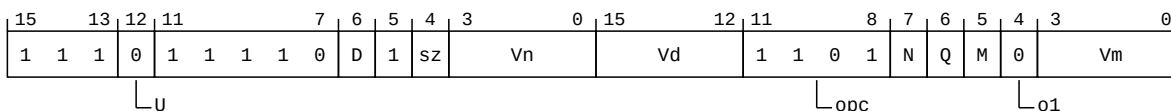
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as `NOP`. This means it behaves as if it fails the Condition code check.

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VSUB{<c>}{<q>},<dt> {<Dd>,<Dn>,<Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VSUB{<c>} {<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if sz == '1' then Undefined(); end;
let advsimd : boolean = TRUE;
let esize : integer{} = 32 >> UInt(sz);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

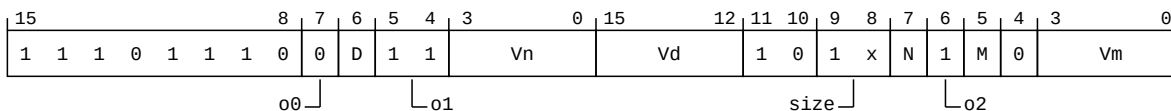
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Encoding for the Single-precision scalar variant

Applies when `(size == 10)`

```
VSUB{<c>}{<q>}.F32 {<Sd>, }<Sn>, <Sm>
```

Encoding for the Double-precision scalar variant

Applies when `(size == 11)`

```
VSUB{<c>}{<q>}.F64 {<Dd>, }<Dn>, <Dm>
```

Decode for all variants of this encoding

```
if FPSCR().Len != '000' || FPSCR().Stride != '00' then Undefined(); end;
if size != '1x' then Undefined(); end;
let advsimd : boolean = FALSE;
let esize : integer{} = 8 << UInt(size);
let d : integer = if size == '11' then UInt(D::Vd) else UInt(Vd::D);
let n : integer = if size == '11' then UInt(N::Vn) else UInt(Vn::N);
let m : integer = if size == '11' then UInt(M::Vm) else UInt(Vm::M);
let regs : integer = ARBITRARY : integer;
let elements : integer = ARBITRARY : integer;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01'` && `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “A2 Double-precision scalar”, “A2 Single-precision scalar”, “T1 128-bit SIMD vector”, “T1 64-bit SIMD vector”, “T2 Double-precision scalar”, and “T2 Single-precision scalar” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

<Sd>

Is the 32-bit name of the SIMD&FP destination register, encoded in the “Vd:D” field.

<Sn>

Is the 32-bit name of the first SIMD&FP source register, encoded in the “Vn:N” field.

<Sm>

Is the 32-bit name of the second SIMD&FP source register, encoded in the “Vm:M” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        let fpcr : FPCR_Type = StandardFPCR();
        for r = 0 to regs-1 do
            for e = 0 to elements-1 do
                D(d+r)[e*:esize] = FPSub{esize}(D(n+r)[e*:esize], D(m+r)[e*:esize], fpcr);
            end;
        end;
    else // VFP instruction
```

```
    let fpcr : FPCR_Type = EffectiveFPCR();  
    case esize of  
        when 32 =>  
            S(d) = FPSub{32}(S(n), S(m), fpcr);  
        when 64 =>  
            D(d) = FPSub{64}(D(n), D(m), fpcr);  
    end;  
end;  
end;
```

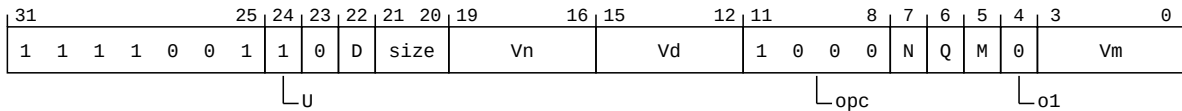
C6.1.223 VSUB (integer)

Vector Subtract (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

`VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>`

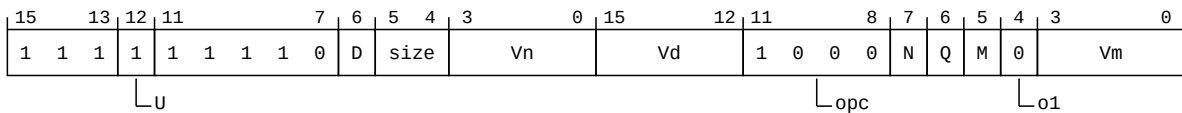
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

`VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            D(d+r)[e*:esize] = D(n+r)[e*:esize] - D(m+r)[e*:esize];
        end;
    end;
end;
```

C6.1.224 VSUBHN

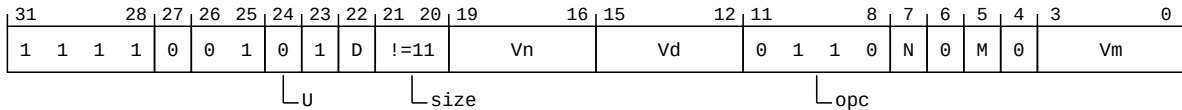
Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. For rounded results, see [VRSUBHN](#).

There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding

VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

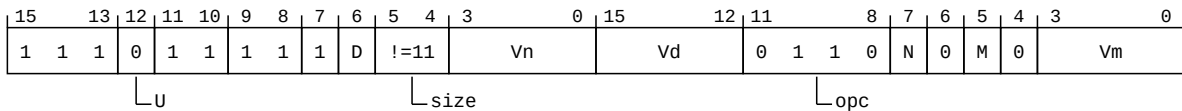
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

T1



Encoding

VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vn[0] == '1' || Vm[0] == '1' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);

```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        let result : bits(2*esize) = Qin(n>>1)[e*(2*esize)] - Qin(m>>1)[e*(2*esize)];
        D(d)[e*esize] = result[2*esize-1:esize];
    end;
end;
```

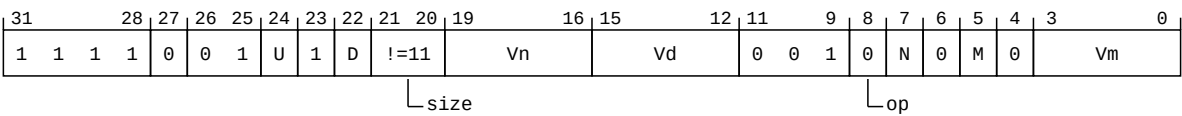
C6.1.225 VSUBL

Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding

VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

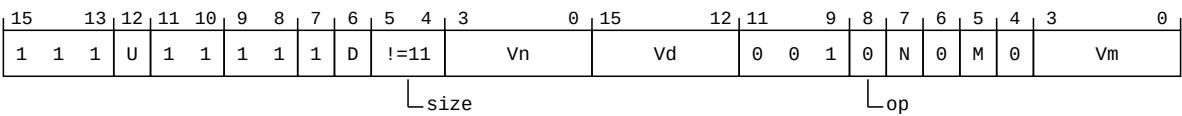
Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

T1



Encoding

VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the second operand vector, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vsubw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:(esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:(esize)];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 - element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

C6.1.226 VSUBW

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	9	8	7	6	5	4	3	0			
1	1	1	1	0	0	1	U	1	D	!=11	Vn		Vd		0	0	1	1	N	0	M	0	Vm		
										size														op	

Encoding

VSUBW{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

T1

15	13	12	11	10	9	8	7	6	5	4	3	0	15	12	11	9	8	7	6	5	4	3	0	
1	1	1	U	1	1	1	1	D	!=11	Vn			Vd			0	0	1	1	N	0	M	0	Vm
										size												op		

Encoding

VSUBW{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm>

Decode for this encoding

```

if size == '11' then See("Related encodings"); end;
if Vd[0] == '1' || (op == '1' && Vn[0] == '1') then Undefined(); end;
let unsigned : boolean = (U == '1');
let is_vsubw : boolean = (op == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let elements : integer = 64 DIV esize;

```

Related encodings: See [Advanced SIMD data-processing](#) for the T32 instruction set, or [Advanced SIMD data-processing](#) for the A32 instruction set.

Assembler Symbols

<c>

For the “A1” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.
For the “T1” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the second operand vector, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for e = 0 to elements-1 do
        var element1 : integer;
        if is_vsubw then
            let op1elt : bits(2*esize) = Qin(n>>1)[e*:(2*esize)];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        else
            let op1elt : bits(esize) = Din(n)[e*:esize];
            element1 = if unsigned then UInt(op1elt) else SInt(op1elt);
        end;
        let op2elt : bits(esize) = Din(m)[e*:esize];
        let element2 : integer = if unsigned then UInt(op2elt) else SInt(op2elt);
        let result : integer = element1 - element2;
        Q(d>>1)[e*:(2*esize)] = result[2*esize-1:0];
    end;
end;
```

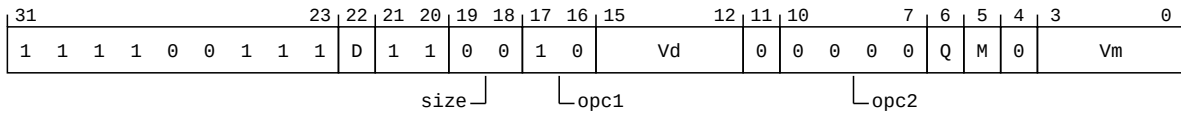
C6.1.227 VSWP

Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when $(Q == 0)$

VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VSWP}\{\langle c \rangle\}\{\langle q \rangle\}\{.\langle dt \rangle\} \quad \langle Qd \rangle, \langle Qm \rangle$$

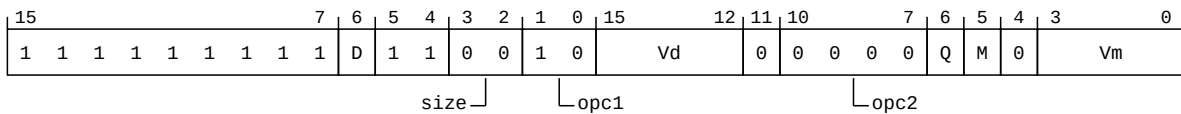
Decode for all variants of this encoding

```

if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when $(Q == 1)$

VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>

Decode for all variants of this encoding

```

if size != '00' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

An optional data type. It is ignored by assemblers, and does not affect the encoding.

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        if d == m then
            D(d+r) = ARBITRARY : bits(64);
        else
            D(d+r) = Din(m+r);
            D(m+r) = Din(d+r);
        end;
    end;
end;

```

C6.1.228 VTBL, VTBX

Vector Table Lookup and Extension

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0						
1	1	1	1	0	0	1	1	1	D	1	1	Vn		Vd		1	0	len	N	op	M	0	Vm	

Encoding for the VTBL variant

Applies when (op == 0)

VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Encoding for the VTBX variant

Applies when (op == 1)

VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Decode for all variants of this encoding

```
let is_vtbl : boolean = (op == '0');
let length : integer = UInt(len)+1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
if n+length > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If n + length > 32, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

T1

15															7	6	5	4	3	0															15	12	11	10	9	8	7	6	5	4	3	0																																																																																																																																			
1															1	1	1	1	1	1	1	1	1	1	D															1	1	Vn															Vd															1															0	len															N															op															M															0															Vm														

Encoding for the VTBL variant

Applies when (`op == 0`)

VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Encoding for the VTBX variant

Applies when ($\text{op} == 1$)

VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>

Decode for all variants of this encoding

```
let is_vtbl : boolean = (op == '0');
let length : integer = UInt(len)+1;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
if n+length > 32 then UnpredictableProcedure(); end;
```

CONSTRAINED UNPREDICTABLE behavior

If $n + \text{length} > 32$, then one of the following behaviors must occur:

- The instruction is `UNDEFINED`.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are `UNKNOWN`. This behavior does not affect any general-purpose registers.

For more information about the **CONSTRAINED UNPREDICTABLE** behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

 $\langle c \rangle$

For the “A1 VTBL” and “A1 VTBX” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 VTBL” and “T1 VTBX” variants: see [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<list>

The vectors containing the table. It must be one of:

{<Dn>} Encoded as len = 0b00.
{<Dn>, <Dn+1>} Encoded as len = 0b01.
{<Dn>, <Dn+1>, <Dn+2>} Encoded as len = 0b10.
{<Dn>, <Dn+1>, <Dn+2>, <Dn+3>} Encoded as len = 0b11.

<Dm>

Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the “M:Vm” field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    let table3 : bits(64) = if length == 4 then D(n+3) else Zeros{64};
    let table2 : bits(64) = if length >= 3 then D(n+2) else Zeros{64};
    let table1 : bits(64) = if length >= 2 then D(n+1) else Zeros{64};
    let table : bits(256) = table3 :: table2 :: table1 :: D(n);

    for i = 0 to 7 do
        let index : integer = UInt(D(m)[i*8]);
        if index < 8*length then
            D(d)[i*8] = table[index*8];
        else
            if is_vtbl then
                D(d)[i*8] = Zeros{8};
            // else D[d]<i*8> unchanged
            end;
        end;
    end;
end;
```

C6.1.229 VTRN

Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

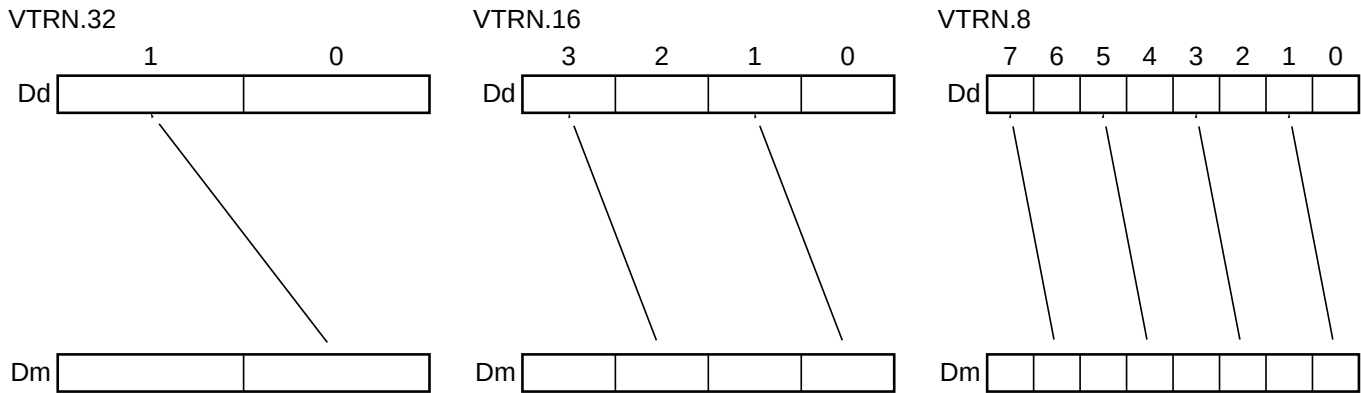


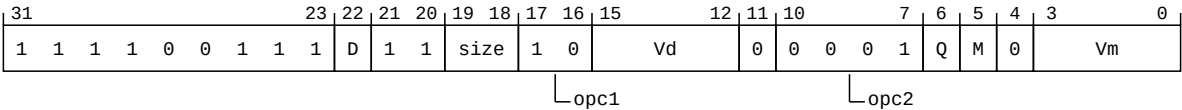
Figure C6-9 VTRN doubleword operations

Depending on settings in the CPACR, NSACR, and HCPTR registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instructions VUZP (alias) and VZIP (alias).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1).

A1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

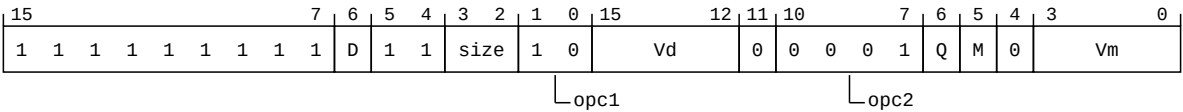
Applies when (Q == 1)

VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>

Decode for all variants of this encoding

```
if size == '11' then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see Standard assembler syntax fields. This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see Standard assembler syntax fields.

<q>

See Standard assembler syntax fields.

<dt>

Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    let h : integer = elements DIV 2;

    for r = 0 to regs-1 do
        if d == m then
            D(d+r) = ARBITRARY : bits(64);
        else
            for e = 0 to h-1 do
                D(d+r)[(2*e+1)*:esize] = Din(m+r)[(2*e)*:esize];
                D(m+r)[(2*e)*:esize] = Din(d+r)[(2*e+1)*:esize];
            end;
        end;
    end;
end;
```

C6.1.230 VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

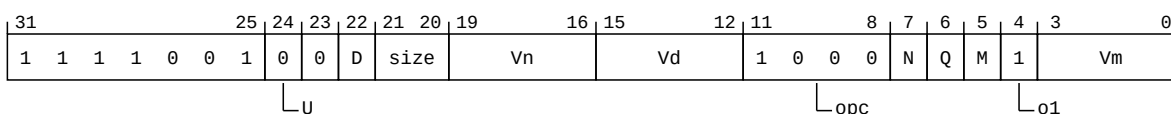
- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be `UNDEFINED`, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VTST{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

VTST{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

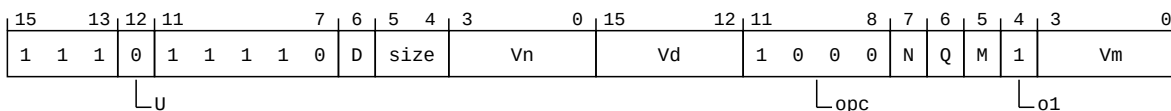
Decode for all variants of this encoding

```

if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VTST{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

VTST{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

Decode for all variants of this encoding

```
if Q == '1' && (Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1') then Undefined(); end;
if size == '11' then Undefined(); end;
let esize : integer{} = 8 << UInt(size);
let elements : integer = 64 DIV esize;
let d : integer = UInt(D::Vd);
let n : integer = UInt(N::Vn);
let m : integer = UInt(M::Vm);
let regs : integer = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	8
01	16
10	32

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dn>

Is the 64-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field.

<Dm>

Is the 64-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qn>

Is the 128-bit name of the first SIMD&FP source register, encoded in the “N:Vn” field as <Qn>*2.

<Qm>

Is the 128-bit name of the second SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    for r = 0 to regs-1 do
        for e = 0 to elements-1 do
            if !IsZero(D(n+r)[e*:esize] AND D(m+r)[e*:esize]) then
                D(d+r)[e*:esize] = Ones(esize);
            else
                D(d+r)[e*:esize] = Zeros(esize);
            end;
        end;
    end;
end;
end;
```

C6.1.231 **VUZP**

Vector Unzip de-interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

VUZP.8, doubleword

	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B6	B4	B2	B0	A6	A4	A2	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	B5	B3	B1	A7	A5	A3	A1

Figure C6-10 VUZP doubleword operation for data type 8

VUZP.32, quadword

	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B2	B0	A2	A0
Qm	B3	B2	B1	B0	B3	B1	A3	A1

Figure C6-11 VUZP quadword operation for data type 32

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31										23		22	21	20	19	18	17	16	15	12			11	10	7			6	5	4	3	0		
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd			0	0	0	1	0	Q	M	0	Vm								
															└─opc1										└─opc2									

Encoding for the 64-bit SIMD vector variant

Applies when ($Q = 0$)

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VUZP}\{\langle c \rangle\} \{\langle q \rangle\} \cdot \langle dt \rangle \quad \langle Q_d \rangle, \quad \langle Q_m \rangle$$

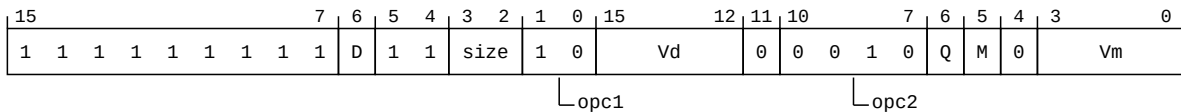
Decode for all variants of this encoding

```

if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D:Vd);
let m : integer = UInt(M:Vm);

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when $(Q == 1)$

$$\text{VUZP}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \langle Qd \rangle, \langle Qm \rangle$$

Decode for all variants of this encoding

```

if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D:Vd);
let m : integer = UInt(M:Vm);

```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

 $\langle q \rangle$

See [Standard assembler syntax fields](#).

<dt>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”;

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q(d>>1) = ARBITRARY : bits(128);
        else
            let zipped_q : bits(256) = Q(m>>1)::Q(d>>1);
            for e = 0 to (128 DIV esize) - 1 do
                Q(d>>1)[e*:esize] = zipped_q[(2*e)*:esize];
                Q(m>>1)[e*:esize] = zipped_q[(2*e+1)*:esize];
            end;
        end;
    else
        if d == m then
            D(d) = ARBITRARY : bits(64);
        else
            let zipped_d : bits(128) = D(m)::D(d);
            for e = 0 to (64 DIV esize) - 1 do
                D(d)[e*:esize] = zipped_d[(2*e)*:esize];
                D(m)[e*:esize] = zipped_d[(2*e+1)*:esize];
            end;
        end;
    end;
end;
end;

```

C6.1.232 VUZP (alias)

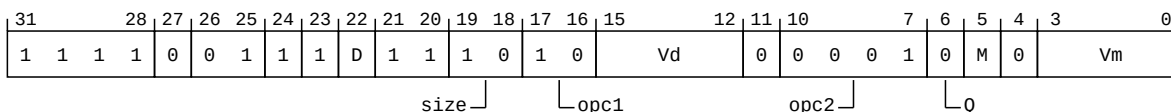
Vector Unzip de-interleaves the elements of two vectors.

This is a pseudo-instruction of [VTRN](#). This means:

- The encodings in this description are named to match the encodings of [VTRN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VTRN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



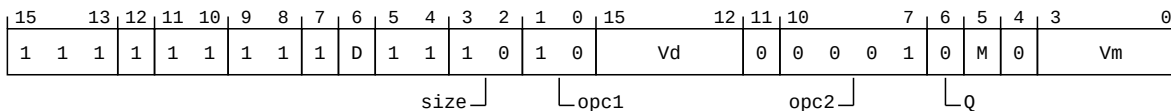
Encoding for the 64-bit SIMD vector variant

VUZP{<c>} {<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>} {<q>}.<dt> <Dd>, <Dm>

T1



Encoding for the 64-bit SIMD vector variant

VUZP{<c>} {<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>} {<q>}.<dt> <Dd>, <Dm>

Assembler Symbols

<c>

For the “A1 64-bit SIMD vector” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 64-bit SIMD vector” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

The description of [VTRN](#) gives the operational pseudocode for this instruction.

C6.1.233 VZIP

Vector Zip interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

VZIP.8, doubleword

	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B3	A3	B2	A2	B1	A1	B0	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	A7	B6	A6	B5	A5	B4	A4

Figure C6-12 VZIP doubleword operation for data type 8

VZIP.32, quadword

	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B1	A1	B0	A0
Qm	B3	B2	B1	B0	B3	A3	B2	A2

Figure C6-13 VZIP quadword operation for data type 32

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31									23		22	21	20	19	18	17	16	15	12			11	10	7			6	5	4	3	0													
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd			0	0	0	1	1	Q	M	0	Vm																		
															└─opc1															└─opc2														

Encoding for the 64-bit SIMD vector variant

Applies when ($Q == 0$)

VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>

Encoding for the 128-bit SIMD vector variant

Applies when ($Q == 1$)

$$\text{VZIP}\{\langle c \rangle\}\{\langle q \rangle\}.\langle dt \rangle \quad \langle Q_d \rangle, \langle Q_m \rangle$$

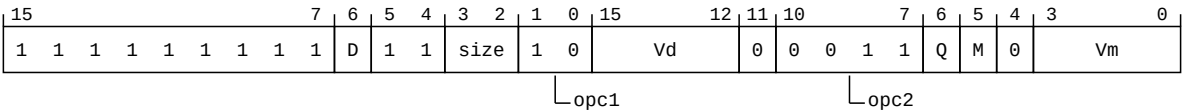
Decode for all variants of this encoding

```

if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D:Vd);
let m : integer = UInt(M:Vm);

```

T1



Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

Decode for all variants of this encoding

```
if size == '11' || (Q == '0' && size == '10') then Undefined(); end;
if Q == '1' && (Vd[0] == '1' || Vm[0] == '1') then Undefined(); end;
let quadword_operation : boolean = (Q == '1');
let esize : integer{} = 8 << UInt(size);
let d : integer = UInt(D::Vd);
let m : integer = UInt(M::Vm);
```

Assembler Symbols

<c>

For the “A1 128-bit SIMD vector” and “A1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 128-bit SIMD vector” and “T1 64-bit SIMD vector” variants: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<dt>

For the “A1 64-bit SIMD vector” and “T1 64-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the “A1 128-bit SIMD vector” and “T1 128-bit SIMD vector” variants: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

<Qd>

Is the 128-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field as <Qd>*2.

<Qm>

Is the 128-bit name of the SIMD&FP source register, encoded in the “M:Vm” field as <Qm>*2.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q(d>>1) = ARBITRARY : bits(128);
        else
            var zipped_q : bits(256);
            for e = 0 to (128 DIV esize) - 1 do
                zipped_q[(2*e)*:esize] = Q(d>>1)[e*:esize];
                zipped_q[(2*e+1)*:esize] = Q(m>>1)[e*:esize];
            end;
            Q(d>>1) = zipped_q[127:0]; Q(m>>1) = zipped_q[255:128];
        end;
    else
        if d == m then
            D(d) = ARBITRARY : bits(64);
        else
            var zipped_d : bits(128);
            for e = 0 to (64 DIV esize) - 1 do
                zipped_d[(2*e)*:esize] = D(d)[e*:esize];
                zipped_d[(2*e+1)*:esize] = D(m)[e*:esize];
            end;
            D(d) = zipped_d[63:0]; D(m) = zipped_d[127:64];
        end;
    end;
end;
end;

```

C6.1.234 VZIP (alias)

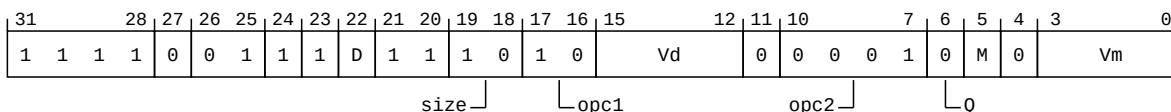
Vector Zip interleaves the elements of two vectors.

This is a pseudo-instruction of [VTRN](#). This means:

- The encodings in this description are named to match the encodings of [VTRN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VTRN](#) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



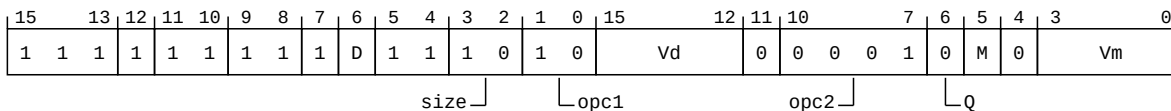
Encoding for the 64-bit SIMD vector variant

VZIP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

T1



Encoding for the 64-bit SIMD vector variant

VZIP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

Assembler Symbols

<c>

For the “A1 64-bit SIMD vector” variant: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For the “T1 64-bit SIMD vector” variant: see [Standard assembler syntax fields](#).

<q>

See [Standard assembler syntax fields](#).

<Dd>

Is the 64-bit name of the SIMD&FP destination register, encoded in the “D:Vd” field.

<Dm>

Is the 64-bit name of the SIMD&FP source register, encoded in the “M:Vm” field.

Operation

The description of [VTRN](#) gives the operational pseudocode for this instruction.

The AArch32 System Level Architecture

Part D

Chapter D1

The AArch32 System Level Programmers' Model

This chapter gives a system level description of the programmers' model for execution in AArch32 state. It contains the following sections:

- [About the AArch32 System level programmers' model](#)
- [Exception levels](#)
- [Exception terminology](#)
- [Execution state](#)
- [Instruction Set state](#)
- [Virtualization](#)
- [AArch32 state PE modes](#)
- [AArch32 general-purpose registers, the PC, and the Special-purpose registers](#)
- [Process state, PSTATE](#)
- [Instruction set states](#)
- [Handling exceptions](#)
- [Routing of aborts](#)
- [Exception return to an Exception level using AArch32](#)
- [Asynchronous exception behavior for exceptions taken from AArch32 state](#)
- [AArch32 state exception descriptions](#)
- [Reset into AArch32 state](#)
- [Mechanisms for entering a low-power state](#)
- [The AArch32 System register interface](#)
- [Advanced SIMD and floating-point support](#)
- [Configurable instruction controls](#)

D1.1 About the AArch32 System level programmers' model

An application programmer has only a restricted view of the system. The System level programmers' model supports this application level view of the system, and includes features that are required for one or both of an operating system (OS) and a hypervisor to provide the programming environment seen by an application. This chapter describes the System level programmers' model when executing at EL1 or higher in an Exception level that is using AArch32.

The system level programmers' model includes all of the system features required to support operating systems and to handle hardware events.

The following sections give a system level introduction to the basic concepts of the Arm architecture AArch32 state, and the terminology that is used for describing the architecture when executing in this state:

- [Exception levels.](#)
- [Exception terminology.](#)
- [Execution state.](#)
- [Instruction Set state.](#)
- [Virtualization.](#)

The rest of this chapter describes the system level programmers' model when executing in AArch32 state.

The other chapters in this part describe:

- The memory system architecture, as seen when executing in an Exception level that is using AArch32:
 - [The AArch32 System Level Memory Model](#) describes the general features of the Armv8 memory model, when executing in AArch32 state, that are not visible at the application level.

Note

[The AArch32 Application Level Memory Model](#) describes the application level view of the memory model.

- The AArch32 System registers, see [AArch32 System Register Descriptions](#).

Note

The T32 and A32 instruction sets include instructions that provide system level functionality, such as returning from an exception. See, for example, ERET.

D1.2 Exception levels

The architecture defines a set of Exception levels, EL0 to EL2, where:

- If EL n is the Exception level, increased values of n indicate increased software execution privilege.
- Execution at EL0 is called *unprivileged execution*.
- EL2 provides support for virtualization.

All implementations must include EL0, EL1 and EL2.

Execution can move between Exception levels only on taking an exception or on returning from an exception:

- On taking an exception, the Exception level can only increase or remain the same.
- On returning from an exception, the Exception level can only decrease or remain the same.

The Exception level that execution changes to or remains in on taking an exception is called the *target Exception level* of the exception.

Each exception type has a target Exception level that is either:

- Implicit in the nature of the exception.
- Defined by configuration bits in the System registers.

An exception cannot target EL0.

D1.2.1 Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

EL0	Applications.
EL1	OS kernel and associated functions that are typically described as <i>privileged</i> .
EL2	Hypervisor.

D1.2.2 Limited use of Privilege level

As described in the [Protected Memory System Architecture](#), a *translation regime* maps a virtual address (VA) to the corresponding physical address (PA). The PMSAv8-32 translation regimes are defined by the Exception levels that use them.

[Table D1-1](#) shows how the PE modes map to these Privilege levels:

Table D1-1 Mapping of PE modes to AArch32 Privilege levels

Privilege level	
PL2	Hyp
PL1	System, FIQ, IRQ, Supervisor, Abort, Undefined
PL0	User

Each privilege level maps to the corresponding Exception level. For example, PL1 maps to EL1.

D1.3 Exception terminology

The following subsections define the terms that are used when describing exceptions:

- [Terminology for taking an exception.](#)
- [Terminology for returning from an exception.](#)
- [Exception levels.](#)
- [Definition of a precise exception.](#)
- [Definitions of synchronous and asynchronous exceptions.](#)

D1.3.1 Terminology for taking an exception

An exception is *generated* when the PE first responds to an exceptional condition. The PE state at this time is the state that the exception is *taken from*. The PE state immediately after taking the exception is the state that the exception is *taken to*.

D1.3.2 Terminology for returning from an exception

To return from an exception, the PE must execute an exception return instruction. The PE state when an exception return instruction is committed for execution is the state the exception *returns from*. The PE state immediately after the execution of that instruction is the state that the exception *returns to*.

D1.3.3 Exception levels

An Exception level, ELn , with a larger value of n than another Exception level, is described as being a *higher* Exception level than the other Exception level. For example, $EL2$ is a higher Exception level than $EL1$.

An Exception level with a smaller value of n than another Exception level is described as being a *lower* Exception level than the other Exception level. For example, $EL0$ is a lower Exception level than $EL1$.

D1.3.4 Definition of a precise exception

An exception is described as *precise* when the exception handler receives the PE state and memory system state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken, and none afterwards.

An exception is described as *imprecise* if it is not precise.

Other than the SError exception all exceptions that are taken are required to be precise. For each occurrence of an SError exception, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

The terms precise and imprecise can also apply to Debug state entry. See [Imprecise entry to Debug state](#).

Where a synchronous exception is generated as part of an instruction that performs more than one single-copy atomic memory access, the definition of precise permits that the values in registers or memory affected by those instructions can be UNKNOWN, provided that:

- The accesses affecting those registers or memory locations do not, themselves, generate exceptions.
- The registers are not involved in the calculation of the memory address that is used by the instruction.

Examples of instructions that perform more than one single-copy atomic memory access are the LDM and STM instructions.

Note

For the definition of a single-copy atomic access, see [Properties of single-copy atomic accesses](#).

D1.3.5 Definitions of synchronous and asynchronous exceptions

An exception is described as *synchronous* if all of the following apply:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The return address presented to the exception handler is guaranteed to indicate the instruction that caused the exception.
- The exception is precise.

An exception is described as *asynchronous* if any of the following apply:

- The exception is not generated as a result of direct execution or attempted execution of the instruction stream.
- The return address presented to the exception handler is not guaranteed to indicate the instruction that caused the exception.
- The exception is imprecise.

For more information about exceptions, see [Handling exceptions](#).

D1.4 Execution state

The Execution state is AArch32, the 32-bit Execution state.

All Exception levels, EL0, EL1 and EL2, *use* AArch32 Execution state. This means that the PE cannot change Execution states at either reset or change of Exception level.

D1.4.1 About the AArch32 PE modes

AArch32 state provides a set of *PE modes* that support normal software execution and handle exceptions. The current mode determines the set of registers that are available, as described in [AArch32 general-purpose registers, the PC, and the Special-purpose registers](#).

The AArch32 modes are:

- Hyp mode. This mode always executes at EL2.
- System, Supervisor, Abort, Undefined, IRQ, and FIQ modes. These modes execute at EL1.
- User mode. This mode always executes at EL0.

For more information on modes, see [AArch32 state PE modes](#).

The mode in use immediately before an exception is taken is described as the mode the exception is *taken from*. The mode that is used on taking the exception is described as the mode the exception is *taken to*.

All of the following define the mode that an exception is taken to:

- The type of exception.
- The mode the exception is taken from.
- Configuration settings defined at EL2.

Hyp mode can create system traps that cause exceptions to EL2. There is an architected hierarchy where EL2 configuration settings affect a common condition, for example interrupt routing. When no traps are enabled for a particular condition, the AArch32 mode an exception is taken to is called the *default mode* for that exception.

A number of different modes can exist at the same Exception level. All modes at a particular Exception level have the same *execution privilege*, meaning they have the same access rights for accesses to memory and to System registers.

D1.5 Instruction Set state

In AArch32 state, the *Instruction Set state* determines the instruction set that the PE is executing. In an implementation that follows the Arm recommendations, the available Instruction Set states are:

T32 state The PE is executing T32 instructions.

A32 state The PE is executing A32 instructions.

Note

In previous versions of the Arm architecture:

- The T32 instruction set was called the Thumb instruction set.
 - The A32 instruction set was called the ARM instruction set.
-

For more information, see [Process state, PSTATE](#).

D1.6 Virtualization

A PE is in *Hyp mode* when it is executing at EL2. An exception return from Hyp mode to software running at EL1 or EL0 is performed using the `ERET` instruction.

EL2 provides a set of features that support virtualizing the state of an R-profile AArch32 implementation. The basic model of a virtualized system involves:

- A hypervisor, running in EL2, that is responsible for switching between *virtual machines*. A virtual machine is comprised of EL1 and EL0.
- A number of Guest operating systems, that each run in EL1, on a virtual machine.
- For each Guest operating system, applications, that usually run in EL0, on a virtual machine.

Note

In some systems, a Guest RTOS is unaware that it is running on a virtual machine, and is unaware of any other Guest RTOS. In other systems, a hypervisor makes the Guest RTOS aware of these facts. The architecture supports both of these models.

The hypervisor assigns a VMID to each virtual machine.

EL2 is implemented to support Guest RTOS management. EL2 provides controls to:

- Provide virtual values for the contents of a small number of identification registers. A read of one of these registers by a Guest RTOS or the applications for a Guest RTOS returns the virtual value.
- *Trap* various operations, including memory management operations and accesses to many other registers. A trapped operation generates an exception that is taken to EL2.
- Route interrupts to the appropriate one of:
 - The current Guest RTOS.
 - A Guest RTOS that is not currently running.
 - The hypervisor.

For memory accesses:

- The implementation provides an independent *translation regime* for memory accesses from EL2.
- For the PL1&0 translation regime, address translation occurs in two stages:
 - Stage 1 maps the *virtual address* (VA) to an *intermediate physical address* (IPA). This is managed at EL1, usually by a Guest RTOS. The Guest RTOS believes that the IPA is the *physical address* (PA).
 - Stage 2 maps the IPA to the PA. This is managed at EL2. The Guest RTOS might be unaware of this stage.

For more information on the translation regimes, see [Protected Memory System Architecture](#).

D1.6.1 The effect of implementing EL2 on the Exception model

An implementation that includes EL2 implements the following exceptions:

- Hypervisor Call (HVC) exception.
- Traps to EL2. [EL2 configurable controls](#), describes these.
- All of the virtual interrupts:
 - Virtual SError.
 - Virtual IRQ.
 - Virtual FIQ.

HVC exceptions are always taken to EL2. All virtual interrupts are always taken to EL1, and can only be taken from EL1 or EL0.

Each of the virtual interrupts can be independently enabled using controls at EL2.

Each of the virtual interrupts has a corresponding physical interrupt. See [Virtual interrupts](#).

When a virtual interrupt is enabled, its corresponding physical exception is taken to EL2. For more information, see [Asynchronous exception behavior for exceptions taken from AArch32 state](#).

EL2 also:

- Provides controls that can be used to route some synchronous exceptions, taken from state, to EL2. For more information, see:
 - [Routing exceptions from EL0 to EL2](#).
 - [Routing debug exceptions to EL2 using AArch32](#).
 - [Routing of aborts taken to AArch32 state](#).
- Provides mechanisms to trap PE operations to EL2. See [EL2 configurable controls](#).
When an operation is trapped to EL2, the hypervisor typically either:
 - Emulates the required operation. The application running in the Guest RTOS is unaware of the trap.
 - Returns an error to the Guest RTOS.

D1.6.1.1 Virtual interrupts

The virtual interrupts have names that correspond to the physical interrupts, as shown in [Table D1-2](#).

Table D1-2 The virtual interrupts

Physical interrupt	Corresponding virtual interrupt
External SError	Virtual SError
IRQ	Virtual IRQ
FIQ	Virtual FIQ

Software executing at EL2 can use virtual interrupts to signal physical interrupts to EL1 and EL0. [Example D1-1](#) shows a usage model for virtual interrupts.

Example D1-1 Virtual interrupt usage model

A usage model is as follows:

1. Software executing at EL2 routes a physical interrupt to EL2.
2. When a physical interrupt of that type occurs, the exception handler executing in EL2 determines whether the interrupt can be handled in EL2 or requires routing to a Guest RTOS in EL1. If the interrupt requires routing to a Guest RTOS:
 - If the Guest RTOS is currently running, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest RTOS.
 - If the Guest RTOS is not currently running, the physical interrupt is marked as pending for the Guest RTOS. When the hypervisor next switches to the virtual machine that is running that Guest RTOS, the hypervisor uses the appropriate virtual interrupt type to signal the physical interrupt to the Guest RTOS.

EL1 and EL0 modes cannot distinguish a virtual interrupt from the corresponding physical interrupt.

For more information, see [Virtual exceptions](#).

D1.7 AArch32 state PE modes

Table D1-3 shows the PE modes defined by the Arm architecture, for execution in AArch32 state. In this table:

- The *PE mode* column gives the name of each mode and the abbreviation used, for example, in the general-purpose register name suffixes used in [AArch32 general-purpose registers, the PC, and the Special-purpose registers](#).
- The *Encoding* column gives the corresponding `PSTATE.M` field.
- The *Exception level* column gives the Exception level at which the mode is implemented, see [Exception levels](#).

Table D1-3 AArch32 PE modes

PE mode		Encoding	Exception level
User	usr	10000	EL0
FIQ	fiq	10001	EL1
IRQ	irq	10010	EL1
Supervisor	svc	10011	EL1
Abort	abt	10111	EL1
Hyp	hyp	11010	EL2
Undefined	und	11011	EL1
System	sys	11111	EL1

Mode changes can be made under software control, or can be caused by an external or internal exception.

D1.7.1 Notes on the AArch32 PE modes

PE modes are defined only in AArch32 state.

This section gives more information about each of the modes, when it is implemented.

User mode

Software executing in User mode executes at EL0. Execution in User mode is sometimes described as unprivileged execution. Application programs normally execute in User mode, and any program executed in User mode:

- Makes only unprivileged accesses to system resources, meaning it cannot access protected system resources.
- Makes only unprivileged access to memory.
- Cannot change mode except by causing an exception, see [Handling exceptions](#).

System mode

System mode is implemented at EL1.

System mode has the same registers available as User mode, and is not entered by any exception.

Supervisor mode

Supervisor mode is implemented at EL1.

Supervisor mode is the default mode to which a Supervisor Call exception is taken. Executing an `svc` (Supervisor Call) instruction generates a Supervisor Call exception.

Abort mode

Abort mode is implemented at EL1.

Abort mode is the default mode to which a Data Abort exception or Prefetch Abort exception is taken.

Undefined mode

Undefined mode is implemented at EL1.

Undefined mode is the default mode to which an instruction-related exception, including any attempt to execute an UNDEFINED instruction, is taken.

FIQ mode

FIQ mode is implemented at EL1.

FIQ mode is the default mode to which an FIQ interrupt is taken.

IRQ mode

IRQ mode is implemented at EL1.

IRQ mode is the default mode to which an IRQ interrupt is taken.

Hyp mode

Hyp mode is the EL2 mode.

Hyp mode is entered on taking an exception that must be taken to EL2.

The PE enters Hyp mode on reset.

The Hypervisor Call exception and Hyp Trap exception are implemented as part of EL2 and are always taken to Hyp mode when EL2 is using AArch32.

Executing an `HVC` (Hypervisor Call) instruction generates a Hypervisor Call exception. See [Hypervisor Call \(HVC\) exception](#).

For more information, see [Hyp mode](#).

D1.7.2 Hyp mode

Hyp mode is the EL2 mode. When EL2 is using AArch32, it provides the usual method of controlling the virtualization of execution at EL1 and EL0.

This section summarizes how Hyp mode differs from the other modes, and references where this part of the manual describes the features of Hyp mode in more detail:

- Software executing in Hyp mode executes at EL2.
- In Non-debug state, the only mechanisms for changing to Hyp mode are:
 - An exception taken from a EL1 or EL0 mode.
- In Hyp mode, the only exception return is execution of an `ERET` instruction, see `ERET`.
- In Hyp mode, the `CPACR` has no effect on the execution of:
 - System register access instructions.
 - Advanced SIMD and floating-point instructions.

The `HCPTTR` controls execution of these instructions in Hyp mode.

- If software running in Hyp mode executes an `SVC` instruction, the Supervisor Call exception generated by the instruction is taken to Hyp mode, see `SVC`.
- The instructions described in the following sections are UNDEFINED if executed in Hyp mode:
 - `SRS`. See [SRS](#), [SRSDA](#), [SRSDDB](#), [SRSIA](#), [SRSIB](#).
 - `RFE`. See [RFE](#), [RFEDA](#), [RFEDB](#), [RFEIA](#), [RFEIB](#).
 - [LDM \(exception return\)](#).
 - [LDM \(User registers\)](#).

- [STM \(User registers\)](#).
- The `SUBS PC, LR` forms of the instructions described in [SUB, SUBS \(immediate\)](#).

Note

In T32 state, `ERET` is encoded as `SUBS PC, LR, #0`, and therefore this is a valid instruction.

- The exception return form of the instructions described in [MOV, MOVS \(register\)](#).

In addition, deprecated forms of the A32 `ADCS`, `ADDS`, `ANDS`, `BICS`, `EORS`, `MOVS`, `MVNS`, `ORRS`, `RSBS`, `RSCS`, `SBCS`, and `SUBS` instructions with the PC as the destination register are UNDEFINED if executed in Hyp mode. The instruction descriptions identify these UNDEFINED cases.

- The Load unprivileged and Store unprivileged instructions `LDRT`, `LDRSHT`, `LDRHT`, `LDRBT`, `STRT`, `STRHT`, and `STRBT`, are CONSTRAINED UNPREDICTABLE if executed in Hyp mode.

If `HCR.HCD` is set to 1, the `HVC` instruction is UNDEFINED.

D1.7.3 Pseudocode description of mode operations

The `BadMode ()` function tests whether a 5-bit mode number corresponds to one of the permitted modes.

The `BadMode ()` function is defined in [R-profile Architecture Pseudocode](#).

D1.8 AArch32 general-purpose registers, the PC, and the Special-purpose registers

The [general-purpose registers, and the PC, in AArch32 state](#) describes the application level view of the general-purpose registers, and the PC. This view provides:

- The general-purpose registers R0-R14, of which:
 - The preferred name for R13 is SP (*stack pointer*).
 - The preferred name for R14 is LR (*link register*).
- The PC, which can be described as R15.

Figure D1-1 shows the full set of banked general-purpose registers, and the Special-purpose registers:

- The Program Status Registers [CPSR](#) and [SPSR](#).
- [ELR_hyp](#).

Note

The architecture uses system level register names, such as R0_usr, R8_usr, and R8_fiq, when it must identify a specific register. The application level names refer to the registers for the current mode, and usually are sufficient to identify a register.

		System level view							
		User	System	Hyp	Supervisor	Abort	Undefined	IRQ	FIQ
Application level view	R0	R0_usr							
	R1	R1_usr							
	R2	R2_usr							
	R3	R3_usr							
	R4	R4_usr							
	R5	R5_usr							
	R6	R6_usr							
	R7	R7_usr							
	R8	R8_usr							R8_fiq
	R9	R9_usr							R9_fiq
	R10	R10_usr							R10_fiq
	R11	R11_usr							R11_fiq
	R12	R12_usr							R12_fiq
	SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_irq	SP_fiq
	LR	LR_usr			LR_svc	LR_abt	LR_und	LR_irq	LR_fiq
	PC	PC_usr							
	APSR	CPSR		SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Cells with no entry indicate that the User mode register is used.

Figure D1-1 AArch32 general-purpose registers, PC, and Special-purpose registers, showing banking

As described in [PE mode for taking exceptions](#), on taking an exception the PE changes mode, unless it is already in the mode to which it must take the exception. Each mode that the PE might enter in this way has:

- A banked copy of the stack pointer, for example `SP_irq` and `SP_hyp`.
- A register that holds a preferred return address for the exception. This is:
 - For the EL2 mode, Hyp mode, the Special-purpose register `ELR_hyp`.
 - For the other privileged modes to which exceptions can be taken, a banked copy of the link register, for example `LR_und` and `LR_mon`.
- A saved copy of `PSTATE`, made on exception entry, for example `SPSR_irq` and `SPSR_hyp`.

In addition, FIQ mode has banked copies of the general-purpose registers R8 to R12.

User mode and System mode share the same general-purpose registers.

User mode, System mode, and Hyp mode share the same LR.

For more information about the application level view of the SP, LR, and PC, and the alternative descriptions of them as R13, R14 and R15, see [The general-purpose registers, and the PC, in AArch32 state](#).

In AArch32 state, the Special-purpose registers are:

- The `CPSR` and its view as the `APSR`.
- The `SPSR`, including the banked copies `SPSR_abt`, `SPSR_fiq`, `SPSR_hyp`, `SPSR_irq`, `SPSR_mon`, `SPSR_svc`, and `SPSR_und`.
- The `ELR_hyp`.

D1.8.1 Pseudocode description of general-purpose register and PC operations

The following pseudocode gives access to the general-purpose registers and the PC. These registers are an array, `_R`, indexed by parameter `n`.

`LookUpRIndex()` looks up the index value, `n`, for the specified register number and PE mode, using `RBankSelect()` to evaluate the result.

`_R` accesses the specified general-purpose register in the current PE mode, using `Rmode()` to access the register, accessing `_R` if necessary. `SP` accesses the stack pointer, `LR` accesses the link register, and `PC32` accesses the Program Counter. Each function has a non-assignment form for register reads and an assignment form for register writes, other than `PC32`, which has only a non-assignment form.

`BranchTo()` performs a branch to the specified address.

The `_R_PC`, `LR`, `PC32`, `SP`, `LookUpRIndex()`, `RBankSelect()`, `Rmode()`, and `BranchTo()` functions are defined in [R-profile Architecture Pseudocode](#).

D1.8.2 Saved Program Status Registers (SPSRs)

The Saved Program Status Registers (SPSRs) are used to save PE state on taking exceptions. In AArch32 state, there is an SPSR for every mode that an exception can be taken to, as shown in [Figure D1-1](#). For example, the SPSR for Hyp mode is called `SPSR_hyp`.

Note

Exceptions cannot be taken to EL0.

When the PE takes an exception, PE state is saved from `PSTATE` in the SPSR for the mode the exception is taken to. For example, if the PE takes an exception to Hyp mode, PE state is saved in `SPSR_hyp`. For more information on `PSTATE`, see [Process state, PSTATE](#).

Note

All `PSTATE` fields are saved, including those which have no direct read and write access.

Saving the `PSTATE` fields means the exception handler can:

- On return from the exception, restore the PE state to the values it had immediately before the exception was taken. When the PE returns from an exception, PE state is restored to the state stored in the SPSR of the mode the exception is returning from, if the exception return is made using one of:
 - ERET.
 - LDM.
 - The Exception return form of the instruction described in [MOV, MOVS \(register\)](#).
 - The Exception return form of the instruction described in [SUB, SUBS \(immediate\)](#).

For example, on returning from Hyp mode, PE state is restored to the state stored in [SPSR_mon](#). If the exception return is made using the RFE instruction, the PE restores the PE state from an SPSR valued read from memory.
- Examine the value that [PSTATE](#) had when the exception was taken, for example to determine the instruction set state and privilege level in which the instruction that caused an Undefined Instruction exception was executed.

The SPSRs are UNKNOWN on a Warm reset. Any operation in an EL1 or EL0 mode makes [SPSR_hyp](#) unknown.

SPSR bits that are defined as RES0 on an exception taken are ignored on any exception return.

For more information on SPSR, see [SPSR](#).

D1.8.2.1 Pseudocode description of SPSR operations

The following pseudocode gives access to the SPSRs.

The `SPSR_ELx []` function accesses the current SPSR.

The `SPSRWriteByInstr ()` function is used by the [MSR \(register\)](#) and [MSR \(immediate\)](#) instructions to update the current SPSR.

The `SPSR_ELx []` and `SPSRWriteByInstr ()` functions are defined in [R-profile Architecture Pseudocode](#).

D1.8.3 ELR_hyp

Hyp mode does not provide its own banked copy of LR. Instead, on taking an exception to Hyp mode, the preferred return address is stored in `ELR_hyp`, a 32-bit Special-purpose register implemented for this purpose.

`ELR_hyp` can be accessed explicitly only by executing:

- An MRS or MSR instruction that targets `ELR_hyp`, see:
 - [MRS \(Banked register\)](#).
 - [MSR \(Banked register\)](#).

The `ERET` instruction uses the value in `ELR_hyp` as the return address for the exception. For more information, see [ERET](#).

Software execution in any EL1 or EL0 mode makes `ELR_hyp` UNKNOWN.

D1.9 Process state, PSTATE

Process state or PSTATE is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

PSTATE is defined in pseudocode as the PSTATE structure, of type ProcState. ProcState is defined in [A-profile Architecture Pseudocode](#).

The PSTATE fields that are meaningful in AArch32 state are:

The Condition flags

N	Negative Condition flag.
Z	Zero Condition flag.
C	Carry Condition flag.
V	Overflow Condition flag.

[Process state, PSTATE](#) gives more information about these.

The overflow or saturation flag

Q	See Process state, PSTATE .
---	---

The greater than or equal flags

GE[3:0]	See Process state, PSTATE .
---------	---

The PE state controls

J, T	Instruction set state. See PSTATE . J is RES0. On a Warm reset, T is set to an IMPLEMENTATION DEFINED value. On taking an exception to: - A PL1 mode, T is set to SCTLR.TE. - EL2, T is set to HSCTLR.TE.
IT[7:0]	IT block state bits. See Process state, PSTATE . On a Warm reset or taking an exception, these bits are set to 0.
E	Endianness of data accesses. See Process state, PSTATE . If an implementation provides both Big-endian and Little-endian support, then: - On a Warm reset this bit is set to the IMPLEMENTATION DEFINED reset value of HSCTLR.EE. - On taking an exception to: - A PL1 mode, this bit is set to SCTLR.EE. - EL2, this bit is set to HSCTLR.EE
IL	Illegal Execution state bit. See The Illegal Execution state exception . On a Warm reset or taking an exception to AArch32 state, this bit is set to 0.

For information on how the J, T, IT[7:0], E, and IL fields can be accessed, see [Accessing the PE state controls and the Execution state bit](#).

The asynchronous exception mask bits

A	SError mask bit.
I	IRQ interrupt mask bit.
F	FIQ interrupt mask bit.

For each bit, the values are:

- 0 Exception not masked.
- 1 Exception masked.

On a Warm reset, these bits are set to 1.

On taking an exception, one or more of these bits are set to 1.

For more information, see both:

- Asynchronous exception masking controls.
- PE state on exception entry.

The mode bits

- M[4:0]** Current mode of the PE. Table D1-3 lists the permitted values of this field. All other values are reserved. Illegal changes to PSTATE.M describes the effect of setting M[4:0] to a reserved value.
- M[4] is:
- M[4], RES1
- On a Warm reset, M[4:0] is set to 0b11010 meaning Hyp mode.
- On taking an exception, M[4:0] is set to the target mode for the exception type.
- For more information about the PE modes, see:
- AArch32 state PE modes.
 - PE state on exception entry.

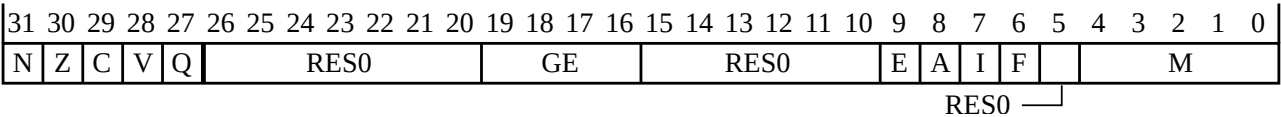
D1.9.1 Accessing PSTATE fields

- The PSTATE fields can be accessed as described in the following subsections:
- The Current Program Status Register, CPSR.
 - Accessing the PE state controls, the Execution state bit, and the Undefined Instruction exception injection bit.
 - The CPS instruction.
 - The SETEND instruction.

D1.9.1.1 The Current Program Status Register, CPSR

Some PSTATE fields can be accessed using the Special-purpose Current Program Status Register (CPSR). The CPSR can be directly read using the MRS instruction, and directly written using the MSR (register) and MSR (immediate) instructions.

The CPSR bit assignments are:



- N, Z, C, V, bits [31:28]** The PSTATE Condition flags.
- Q, bit [27]** The PSTATE overflow or saturation flag.

Bits[26:20, 15:10, 5]

Reserved, RES0.

GE[3:0], bits [19:16]

The [PSTATE](#) greater than or equal flags.

E, bit [9] The [PSTATE](#) endianness bit.

A, I, F, bits [8:6]

The [PSTATE](#) asynchronous exception mask bits.

M[4:0], bits [4:0]

The [PSTATE](#) mode bits.

The other [PSTATE](#) fields cannot be accessed by using the CPSR. For information on how to access them, see [Accessing the PE state controls, the Execution state bit, and the Undefined Instruction exception injection bit](#).

The application level alias for the CPSR is the [APSR](#). The [APSR](#) is a subset of the CPSR. See [The Application Program Status Register, APSR](#).

Writes to the CPSR have side-effects on various aspects of PE operation. All of these side-effects, except side-effects on memory accesses associated with fetching instructions, are synchronous to the CPSR write. This means that they are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

The privilege level and address space of memory accesses associated with fetching instructions depend on the current Exception level and Security state. Writes to [PSTATE.M](#) can change one or both of the Exception level and Security state. The effect, on memory accesses associated with fetching instructions, of a change of Exception level or Security state is:

- Synchronous to the change of Exception level or Security state, if that change is caused by an exception entry or exception return.
- Guaranteed not to be visible to any memory access caused by fetching an earlier instruction in the execution stream.
- Guaranteed to be visible to any memory access caused by fetching any instruction after the next [Context Synchronization event](#) in the execution stream.
- Might or might not affect memory accesses caused by fetching instructions between the mode change instruction and the point where the mode change is guaranteed to be visible.

See [Exception return](#) for the definition of exception return instructions.

D1.9.1.2 Accessing the PE state controls, the Execution state bit, and the Undefined Instruction exception injection bit

The PE state controls are the [PSTATE](#).{IL, IT[7:0], J, E, T} fields. Software can read or write these in an [SPSR](#).

In the [CPSR](#):

- The PE state controls, other than [PSTATE.E](#), are RAZ when read by an MRS instruction.
- Writes to the PE state controls, other than [PSTATE.E](#), by [MSR \(register\)](#) or [MSR \(immediate\)](#), are ignored in all modes.

Instructions other than MRS, [MSR \(register\)](#), or [MSR \(immediate\)](#) that access the PE state controls can read and write them in any mode.

Unlike the other [PSTATE](#) PE state controls, [PSTATE.E](#) can be read by an MRS instruction and might be written by [MSR \(register\)](#) or [MSR \(immediate\)](#). However, Arm deprecates [PSTATE.E](#) having a different value from the equivalent System register EE bit, see [Mixed-endian support in AArch32](#).

Note

To determine the current endianness, software can use an `LDR` instruction to load a word from memory with a known value that differs if the endianness is reversed. For example, using an `LDR` instruction to load a word whose four bytes are `0x01`, `0x00`, `0x00`, and `0x00` in ascending order of memory address loads the destination register with:

- `0x0000_0001` if the current endianness is little-endian.
 - `0x0100_0000` if the current endianness is big-endian.
-

The `PSTATE.M[4]` bit is the Execution state bit. When read by an `MRS` instruction in AArch32 state, this bit always reads as 1. When written by an `MSR (register)` instruction or `MSR (immediate)` instruction, writing a value other than 1 is an illegal change to the `PSTATE.M` field. See [Illegal changes to PSTATE.M](#).

D1.9.1.3 The CPS instruction

The T32 and A32 instruction sets both include an instruction to manipulate `PSTATE.{A, I, F}` and `PSTATE.M`:

`CPSIE <iflags> {, #<mode>}` Sets the specified `PSTATE.{A, I, F}` exception masks to 0, enabling the exception, and optionally changes to the specified mode.

`CPSID <iflags> {, #<mode>}` Sets the specified `PSTATE.{A, I, F}` exception masks to 1, disabling the exception, and optionally changes to the specified mode.

`CPS #<mode>` Changes to the specified mode without affecting the `PSTATE.{A, I, F}` exception masks.

The `CPS` instruction is unconditional. For more information, see [CPS](#), [CPSID](#), [CPSIE](#).

D1.9.1.4 The SETEND instruction

The T32 and A32 instruction sets both include an instruction to manipulate `PSTATE.E`:

`SETEND BE` Sets `PSTATE.E` to 1, for big-endian operation.

`SETEND LE` Sets `PSTATE.E` to 0, for little-endian operation.

The `SETEND` instruction is unconditional. For more information, see [SETEND](#). Arm deprecates use of the `SETEND` instruction.

D1.9.2 The Saved Program Status Registers (SPSRs)

On taking an exception, `PSTATE` is preserved in the SPSR of the mode to which the exception is taken. The SPSRs are described in [Saved Program Status Registers \(SPSRs\)](#).

D1.9.3 Illegal changes to PSTATE.M

In PE modes other than User mode, `MSR` and `CPS` instructions can explicitly change `PSTATE.M`. The following changes to `PSTATE.M` by `MSR` or `CPS` instructions are illegal:

- A change to an encoding that [Table D1-3](#) does not show.
- A change to a mode that is not implemented.
- A change to a mode that is not accessible from the context the `MRS` or `CPS` instruction is executed in, as follows:
 - A change to a mode that would cause entry to a higher Exception level.
 - A change to Hyp mode from any other mode.
 - A change from Hyp mode to any other mode.
 - When the value of `HCR.TGE` is 1, attempting to change from Hyp mode to a PL1 mode, see [Trapping of general exceptions to Hyp mode](#).

On executing an instruction that attempts an illegal change to `PSTATE.M`:

- `PSTATE.M` is unchanged, and the current mode remains unchanged.

- [PSTATE.IL](#) is set to 1.
- All other [PSTATE](#) fields are written to as normal.

———— **Note** ————

For the [PSTATE](#) fields that MSR and CPS instructions update, see the instruction descriptions:

- [MSR \(register\)](#).
 - [MSR \(immediate\)](#).
 - [CPS](#), [CPSID](#), [CPSIE](#).
-

When the value of [PSTATE.IL](#) is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See [The Illegal Execution state exception](#).

———— **Note** ————

The PE ignores writes to [PSTATE.M](#) when executing at PL0.

D1.9.4 Pseudocode description of PSTATE operations

The [CPSRWriteByInstr\(\)](#) function is used by the [MSR \(register\)](#) and [MSR \(immediate\)](#) instructions to update [PSTATE](#).

The [SetPSTATEFromPSR\(\)](#) function updates [PSTATE](#) from a CPSR or SPSR.

[R-profile Architecture Pseudocode](#) defines these functions.

D1.10 Instruction set states

The instruction set states are described in [The AArch32 Application Level Memory Model](#) and application level operations on them are described there. This section supplies more information about how they interact with system level functionality, in the sections:

- [Exceptions and instruction set state.](#)
- [Unimplemented instruction sets.](#)

D1.10.1 Exceptions and instruction set state

If an exception is taken to an EL1 mode, the [SCTLR.TE](#) bit for the Security state the exception is taken to determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

If the exception is taken to Hyp mode, the [HSCTLR.TE](#) bit determines the instruction set state that handles the exception, and if necessary, the PE changes to this instruction set state on exception entry.

On coming out of reset, the PE starts execution in Hyp mode, in the instruction set state determined by the reset value of [HSCTLR.TE](#).

For more information about exception entry, see [Overview of exception entry](#).

D1.10.2 Unimplemented instruction sets

The [PSTATE.T](#) bit defines the current instruction set state, see [Process state, PSTATE](#).

From the introduction of the Armv8 architecture, there is no support for the hardware acceleration of Java bytecodes, and the Jazelle Instruction set state is obsolete. Every AArch32 implementation must support the Trivial Jazelle implementation described in [Trivial implementation of the Jazelle extension](#).

Note

In previous versions of the Arm architecture, the [PSTATE.{J, T}](#) bits determined the Instruction set state. From the introduction of Armv8, [PSTATE.J](#) is RES0.

D1.10.2.1 Trivial implementation of the Jazelle extension

The implementation of AArch32 state is required to include the trivial Jazelle implementation.

In a trivial implementation of the Jazelle extension:

- At EL1 or EL2, if the Exception level is using AArch32:
 - The [JMCR](#) and [JOSCR](#) are RAZ/WI.
 - The [JIDR](#) is a RAZ read-only register.
- At EL0 when EL0 is using AArch32:
 - It is IMPLEMENTATION DEFINED whether the [JMCR](#) and [JOSCR](#) are RAZ/WI or UNDEFINED.
 - It is IMPLEMENTATION DEFINED whether [JIDR](#) is RAZ or UNDEFINED.
- The [BXJ](#) instruction behaves identically to the [BX](#) instruction in all circumstances.

Note

This is consistent with the [JMCR.JE](#) bit being RAZ, and means that the T32 and A32 instruction sets do not provide any mechanism for attempting to enter Jazelle state.

- Jazelle state, as defined in previous versions of the Arm architecture, is an unimplemented instruction set state.

These requirements ensure that operating systems that support an EJVM execute correctly.

A trivial implementation is not required to extend the PC to 32 bits, that is, it can implement PC[0] as RAZ/WI.

Note

This is because the only way that PC[0] is visible in A32 or T32 state is as a result of an exception occurring during Jazelle state execution, and Jazelle state execution cannot occur on a trivial implementation.

D1.11 Handling exceptions

An exception causes the PE to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal and external sources.

Normally, when an exception is taken the PE state is preserved immediately, before handling the exception. This means that, when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken.

More than one exception might be generated at the same time, and a new exception can be generated while the PE is handling an exception.

The following sections describe exception handling:

- [Exception vectors and the exception base address.](#)
- [Exception prioritization for exceptions taken to AArch32 state.](#)
- [Overview of exception entry.](#)
- [PE mode for taking exceptions.](#)
- [PE state on exception entry.](#)
- [Routing exceptions from EL0 to EL2.](#)
- [Routing debug exceptions to EL2 using AArch32.](#)

See also:

- [Routing of aborts taken to AArch32 state.](#)
- [Exception return to an Exception level using AArch32.](#)
- [Asynchronous exception behavior for exceptions taken from AArch32 state.](#)
- [AArch32 state exception descriptions.](#)

D1.11.1 Exception vectors and the exception base address

When an exception is taken, PE execution is forced to an address that corresponds to the type of exception. This address is called the *exception vector* for that exception. The vectors for the different types of exception form a *vector table*.

A set of exception vectors for an Exception level that is using AArch32 comprises eight consecutive word-aligned memory addresses, starting at an *exception base address*. These eight vectors form an AArch32 *vector table*.

The number of possible exception base addresses, and therefore the number of vector tables, depends on the implemented Exception levels, as follows:

EL1 vector tables

An implementation must include the following AArch32 vector table:

- An exception table for exceptions taken to EL1 modes other than System mode. This is the EL1 vector table, and is in the address space of the PL1&0 translation regime.

Note

Exceptions cannot be taken to System mode.

EL2 vector tables

An implementation that must include the following additional AArch32 vector table if EL2:

- An exception table for exceptions taken to Hyp mode. This is the Hyp vector table, and is in the address space of the PL2 translation regime.
For this vector table, [HVBAR](#) holds the exception base address.

The following subsections give more information:

- [The vector tables and exception offsets.](#)
- [Pseudocode determination of the exception base address.](#)

D1.11.1.1 The vector tables and exception offsets

Table D1-4 defines the AArch32 vector table entries. The *Hyp* column defines the vector table entries for exceptions taken to Hyp mode.

Exceptions cannot be taken to System mode or User mode.

For more information about determining the mode to which an exception is taken, see [PE mode for taking exceptions](#).

When EL2 is using AArch32, it provides a number of additional exceptions, some of which are not shown explicitly in the vector tables. For more information, see [Offsets of AArch32 exceptions provided by EL2](#).

Table D1-4 The AArch32 vector tables

Offset	Vector tables	
	EL2/Hyp	EL1
0x00	Not used	Not used
0x04	Undefined Instruction, from Hyp mode	Undefined Instruction
0x08	Hypervisor Call, from Hyp mode	Supervisor Call
0x0C	Prefetch Abort, from Hyp mode	Prefetch Abort
0x10	Data Abort, from Hyp mode	Data Abort
0x14	Hyp Trap, or Hyp mode entry ^a	Not used
0x18	IRQ interrupt	IRQ interrupt
0x1C	FIQ interrupt	FIQ interrupt

a. See [Use of offset 0x14 in the Hyp vector table](#).

Table D1-5 Modes for taking the exceptions shown in the vector table

Exception	Mode taken to
Undefined Instruction	Undefined
Supervisor Call	Supervisor
Prefetch Abort	Abort
Data Abort	Abort
IRQ interrupt	IRQ
FIQ interrupt	FIQ

For more information about use of the vector tables, see [Overview of exception entry](#).

D1.11.1.1.1 Offsets of AArch32 exceptions provided by EL2

EL2 provides the following exceptions. When EL2 is using AArch32, these exceptions are taken to Hyp mode, and the PE enters the handlers for these exceptions using the following vector table entries shown in [Table D1-4](#):

Hypervisor Call

If taken from Hyp mode, shown explicitly in the Hyp mode vector table. Otherwise, see [Use of offset 0x14 in the Hyp vector table](#).

Hyp Trap

Shown explicitly in the Hyp mode vector table.

Virtual Abort Entered through the Data Abort vector in the vector table.

Virtual IRQ Entered through the IRQ vector in the vector table.

Virtual FIQ Entered through the FIQ vector in the vector table.

Note

[Virtual exceptions when an implementation includes EL2](#) gives more information about the virtual exceptions.

D1.11.1.1.2 Use of offset 0×14 in the Hyp vector table

The vector at offset 0×14 in the Hyp vector table is used for all exceptions that cause entry to Hyp mode from EL0 and EL1, except for IRQ and FIQ exceptions.

Note

Virtual exceptions are never taken to Hyp mode.

D1.11.1.1.3 Pseudocode determination of the exception base address

For an exception taken to a PL1 mode, the `ExcVectorBase()` function determines the exception base address.

The `ExcVectorBase()` function is defined in [R-profile Architecture Pseudocode](#).

Note

The PL1 modes to which exceptions can be taken are Supervisor mode, Undefined mode, Abort mode, IRQ mode, and FIQ mode.

D1.11.2 Exception prioritization for exceptions taken to AArch32 state

The following sections describe the requirements for the prioritization of synchronous exceptions, and the limits on when asynchronous exceptions can be taken:

- [Synchronous exception prioritization for exceptions taken to AArch32 state](#).
- [Architectural requirements for taking asynchronous exceptions](#).

See also:

- [Debug state entry and debug event prioritization](#) for information about the relative prioritization of exceptions and the debug events that cause entry to Debug state.

D1.11.2.1 Synchronous exception prioritization for exceptions taken to AArch32 state

In principle, any single instruction can generate a number of different synchronous exceptions, between the fetching of the instruction, its decode, and eventual execution. This section describes the prioritization of such exceptions when they are taken to an Exception level that is using AArch32.

Exceptions are prioritized as follows, where 1 is the highest priority.

1-5 These priority numbers are used by AArch64 exceptions or debug events.

6 PC alignment fault exceptions. A PC alignment fault exception can only be taken to an Exception level that is using AArch32 as a result of:

- The CONSTRAINED UNPREDICTABLE handling of a branch to an unaligned address, see [Branching to an unaligned PC](#).
- Exiting from Debug state to AArch32 specifying an unaligned PC value, see [Exiting Debug state](#).

A PC alignment fault exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, see [Prefetch Abort exception reporting a PC alignment fault exception](#).

- 7 Illegal Execution state exceptions. It is IMPLEMENTATION DEFINED whether this is prioritized here or at 12. See [The Illegal Execution state exception](#).
- 8 Prefetch Abort exceptions, that are not covered by priorities 9 or 11. See [Prefetch Abort exception](#) and [AArch32 state prioritization of synchronous aborts from a single stage of address translation](#).
- 9 If prioritized here, Prefetch Abort exceptions due to Synchronous External abort, not on translation table walk or hardware update of translation table.
- 10 Breakpoint exceptions or Address Matching Vector Catch exceptions. See:
 - [Breakpoint exceptions](#).
 - [Vector Catch exceptions](#).

Note

An Exception Trapping Vector Catch exception is generated on exception entry for an exception that has been prioritized as described in this section. This means that it does not have its own entry in this list.

- 11 The exceptions listed for priority 9, if they are prioritized as 11.
- 12 Illegal Execution state exceptions. It is IMPLEMENTATION DEFINED whether this is prioritized here or at 7. See [The Illegal Execution state exception](#).
- 13 This priority number is used by an AArch64 exception.
- 14 Software Breakpoint Exceptions caused by the execution of a BKPT Exception generating instruction.
- 15-18 These priority numbers are used by AArch64 exceptions.
- 19 Exceptions taken from EL1 to EL2 because of one of the following configuration settings:
 - [HSTR.Tn](#).
 - [HCR.TIDCP](#).
- 20 UNDEFINED Instruction exceptions that occur as a result of one or more of the following:
 - An attempt to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation.
 - An attempt to execute an instruction that is defined never to be accessible at the current Exception level regardless of any enables or traps.
 - Debug state execution of an instruction encoding that is not accessible in Debug state.
 - Non-debug state execution of an instruction encoding that is not accessible in Non-debug state.
 - Execution of an HVC instruction when HVC instructions are disabled by [HCR.HCD](#).
 - Execution of an HLT instruction when HLT instructions are disabled by [EDSCR.HDE](#) or when halting is prohibited.
 - In Debug state:
 - Execution of a DCPS1 instruction in EL0 when [HCR.TGE](#) is 1.
 - Execution of a DCPS2 instruction in EL1 or EL0.
 - Execution of a DCPS3 instruction.
 - Execution of an instruction that is UNDEFINED as a result of any of:
 - Being in an IT block when [SCTLR.ITD](#) is 1, or when [HSCTLR.ITD](#) is 1.
 - Executing a SETEND instruction when [SCTLR.SED](#) is 1, or when [HSCTLR.SED](#) is 1.
 - Executing a CP15DMB, CP15DSB, or CP15ISB barrier instruction when [SCTLR.CP15BEN](#) is 0, or when [HSCTLR.CP15BEN](#) is 0.
 - Execution of an instruction that is UNDEFINED because at least one of [FPSCR](#).{Stride, Len} is nonzero, when programming these bits to nonzero values is supported. See [Floating-point exceptions and exception traps](#).
- 21 Reserved.
- 22 Exceptions taken to EL1, or taken to EL2 because the value of [HCR.TGE](#) is 1, that are generated because of configurable access to instructions, and that are not covered by any of priorities 6-20.

23	Exceptions taken from EL0 to EL2 because of one of the following configuration settings: • HSTR.Tn. • HCR.TIDCP.
24	Reserved.
25	Exceptions taken to EL2 because of configuration settings in the HCPTR .
26	Exceptions taken to EL2 because of one of the following configuration settings: • Any setting in HCR, other than the TIDCP bit. • Any setting in CNTHCTL. • Any setting in HDCR.
27	Exceptions taken to EL2 because of configurable access to instructions, and that are not covered by any of priorities 6-26.
28	Exceptions caused by the SMC instruction being UNDEFINED.
29	Exceptions caused by the execution of an Exception generating instruction, SVC, HVC, or SMC.
30-34	Reserved.
35-39	Reserved.
40	Trapped floating-point exceptions, if supported. See Floating-point exceptions and exception traps .
41-44	These priority numbers are used by debug events.
45	In descending priority order: a) Data Abort exceptions on address translations and memory protection region updates. b) Data Abort exceptions due to synchronous External aborts on address translations and memory protection region updates. c) Data Abort exceptions arising from an MPU fault not on a translation, that are not covered by priorities 46 or 48. See Data Abort exception and AArch32 state prioritization of synchronous aborts from a single stage of address translation.
46	Data Abort exceptions due to synchronous External aborts on the final physical address of the address translation process.
	Whether these are prioritized here or as priority 48 is IMPLEMENTATION DEFINED.
	See External aborts .
47	Watchpoint exceptions. See Watchpoint exceptions .
48	The exceptions listed for priority 46 if they are prioritized as 48.
49	Reserved.

For priorities 45-48, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

D1.11.2.2 Architectural requirements for taking asynchronous exceptions

The Arm architecture does not define when asynchronous exceptions are taken. The prioritization of asynchronous exceptions, including virtual asynchronous exceptions, is IMPLEMENTATION DEFINED.

An asynchronous exception that is pending before a [Context Synchronization event](#) in the following list, is taken before the first instruction after the context synchronization event, provided that the pending asynchronous event is not masked:

- Execution of an `ISB` instruction that does not fail its Condition code check.
- Exception entry.
- Exception return.
- Exit from Debug state.

Note

- If the first instruction after the context synchronization event generates a synchronous exception, then the architecture does not define the order in which that synchronous exception and the asynchronous exception are taken.
 - The `ISR` identifies any pending asynchronous exceptions.
 - Interrupts are masked when the PE is in Debug state, and therefore this list of context synchronization events does not include the `DCPS` and `DRPS` instructions.
-

In the absence of a specific requirement to take an asynchronous exception, the only requirement of the architecture is that an unmasked asynchronous exception is taken in finite time.

Note

The taking of an unmasked asynchronous exception in finite time must occur with all code sequences, including with a sequence that consists of unconditional loops.

If an unmasked interrupt was pending but is changed to not pending before it is taken, then the architecture permits the interrupt to be taken, but does not require this to happen. If the interrupt is taken, then it must be taken before the first [Context Synchronization event](#) after the interrupt was changed to not pending.

`PSTATE` includes a mask bit for each type of asynchronous exception. Setting one of these bits to 1 can prevent the corresponding asynchronous exception from being taken, although other controls can modify the effect of these bits. For more information, see [Asynchronous exception behavior for exceptions taken from AArch32 state](#).

Taking an exception sets an exception-dependent subset of these mask bits.

Note

In some contexts, the `PSTATE.{A, I, F}` bits mask the taking of asynchronous exceptions. The way these are set on exception entry, described in [PSTATE.{A, I, F, M} values on exception entry](#), can prevent an exception handler being interrupted by an asynchronous exception.

D1.11.3 Overview of exception entry

There are some significant differences between the handling of exceptions taken to Hyp mode and exceptions taken to other modes. Because Hyp mode is the EL2 mode, this means that the following descriptions sometimes distinguish between the *EL2 mode* and the *non-EL2 modes*.

On taking an exception to an Exception level that is using AArch32:

1. The hardware determines the mode to which the exception must be taken, see [PE mode for taking exceptions](#).
2. A link value, indicating the *preferred return address* for the exception, is saved. This is a possible return address for the exception handler, and depends on:
 - The exception type.
 - Whether the exception is taken to the EL2 mode or to a non-EL2 mode.
 - For some exceptions taken to non-EL2 modes, the instruction set state when the exception was taken.

Where the link value is saved depends on whether the exception is taken to the EL2 mode.

For more information, see [Link values saved on exception entry](#).

3. The value of `PSTATE` is saved in the `SPSR` for the mode to which the exception must be taken. The value saved in `SPSR.IT[7:0]` is always correct for the preferred return address.
4. `PSTATE` is updated with new context information for the exception handler. This includes:
 - Setting `PSTATE.M` to the PE mode to which the exception is taken.

- Setting the appropriate [PSTATE](#) mask bits. This can disable the corresponding exceptions, preventing uncontrolled nesting of exception handlers.
- Setting the instruction set state to the state required for exception entry.
- Setting the endianness to the required value for exception entry.
- Clearing the [PSTATE.IT\[7:0\]](#) bits to 0.

For more information, see [PE state on exception entry](#).

5. The appropriate exception vector is loaded into the PC, see [Exception vectors and the exception base address](#).
6. Execution continues from the address held in the PC.

For an exception taken to a non-EL2 mode, on exception entry, the exception handler can use the [SRS](#) instruction to store the return state onto the stack of any mode at the same Exception level and in the same Security state, and can use the [CPS](#) instruction to change mode. For more information about the instructions, see [SRS](#), [SRSDA](#), [SRSDB](#), [SRSIA](#), [SRSIB](#) and [CPS](#), [CPSID](#), [CPSIE](#).

Later sections of this chapter describe each of the possible exceptions, and each of these descriptions includes a pseudocode description of the PE state changes on taking that exception. [Table D1-6](#) gives an index to these descriptions:

Table D1-6 Pseudocode descriptions of exception entry for exceptions taken to AArch32 state

Exception	Description of exception entry
Reset	Pseudocode descriptions of reset
Undefined Instruction	Pseudocode description of taking the Undefined Instruction exception
Hyp Trap	Pseudocode description of taking the Hyp Trap exception
Supervisor Call	Pseudocode description of taking the Supervisor Call exception
Hypervisor Call	Pseudocode description of taking the Hypervisor Call exception
Prefetch Abort	Pseudocode description of taking the Prefetch Abort exception
Data Abort	Pseudocode description of taking the Data Abort exception
Virtual Abort	Pseudocode description of taking the Virtual SError interrupt exception
IRQ	Pseudocode description of taking the physical IRQ exception
Virtual IRQ	Pseudocode description of taking the Virtual IRQ exception
FIQ	Pseudocode description of taking the FIQ exception
Virtual FIQ	Pseudocode description of taking the Virtual FIQ exception

The following sections give more information about the PE state changes, for different architecture implementations. However, you must refer to the pseudocode for a full description of the state changes:

- [PE mode for taking exceptions](#).
- [PE state on exception entry](#).

D1.11.3.1 Link values saved on exception entry

On exception entry, a link value for use on return from the exception, is saved. This link value is based on the *preferred return address for the exception*, as shown in [Table D1-7](#):

Table D1-7 Exception return addresses for exceptions taken to AArch32 state

Exception	Preferred return address	Taken to a mode at
Undefined Instruction	Address of the UNDEFINED instruction	EL1 or EL2
Hyp Trap	Address of the trapped instruction	EL2 only
Supervisor Call	Address of the instruction after the SVC instruction	EL1 or EL2
Hypervisor Call	Address of the instruction after the HVC instruction	EL2 only
Prefetch Abort	Address of aborted instruction fetch	EL1 or EL2
Data Abort	Address of instruction that generated the abort	EL1 or EL2
Virtual Abort	Address of next instruction to execute	EL1
IRQ or FIQ	Address of next instruction to execute	EL1 or EL2
Virtual IRQ or Virtual FIQ	Address of next instruction to execute	EL1

———— **Note** ————

- Although Reset is described as an exception, it differs significantly from other exceptions. The architecture has no concept of a return from a Reset and therefore it is not listed in this section.
- For each exception, the preferred return address is not affected by the Exception level from which the exception was taken.

The link value saved, and where it is saved, depend on whether the exception is taken to a non-EL2 mode, or to an EL2 mode, as follows:

Exception taken to a non-EL2 mode

The link value is saved in the LR for the mode to which the exception is taken.

The saved link value is the preferred return address for the exception, plus an offset that depends on the instruction set state when the exception was taken, as [Table D1-8](#) shows:

Table D1-8 Offsets applied to Link value for exceptions taken to non-EL2 modes

Exception	Offset, for PE state of:	
	A32	T32
Undefined Instruction	+4	+2
Supervisor Call	None	None
Prefetch Abort	+4	+4
Data Abort	+8	+8
Virtual Abort	+8	+8
IRQ or FIQ	+4	+4
Virtual IRQ or Virtual FIQ	+4	+4

Exception taken to an EL2 mode

The link value is saved in the [ELR_hyp](#) Special-purpose register.

The saved link value is the preferred return address for the exception, as shown in [Table D1-7](#), with no offset.

D1.11.4 PE mode for taking exceptions

The following principles determine the Exception level to which an exception is taken, and if that Exception level is using AArch32, the PE mode to which the exception is taken:

- An exception cannot be taken to the EL0 mode.
- An exception is taken either:
 - To the Exception level at which the PE was executing when it took the exception.
 - To a higher Exception level.
- Configuration options and other features provided by EL2 can determine the mode to which some exceptions are taken, as follows:
 - If the exception is taken from Hyp mode, it is taken to Hyp mode.
 - Otherwise, the exception is either taken to Hyp mode, as described in [Exceptions taken to Hyp mode](#), or taken to the default mode for the exception.

[Table D1-5](#) shows the default mode to which each exception is taken.

[Asynchronous exception routing controls](#) describes the exception routing controls provided by EL2.

[Routing of aborts taken to AArch32 state](#) gives more information about the modes to which memory aborts are taken.

D1.11.4.1 Exceptions taken to Hyp mode

The following exceptions are taken to EL2:

- Any exception taken from Hyp mode is taken to Hyp mode.
- The following exceptions are taken to Hyp mode:
 - An abort that [Routing of aborts taken to AArch32 state](#) identifies as taken to Hyp mode.
 - A Hyp Trap exception, see [EL2 configurable controls](#).
 - A Hypervisor Call exception. This is generated by executing an HVC instruction.
 - An SError exception, IRQ exception or FIQ exception that is routed to Hyp mode, as described in [Asynchronous exception routing controls](#).
 - A synchronous External abort, Alignment fault, Undefined Instruction exception, or Supervisor Call exception taken from the EL0 mode and explicitly routed to Hyp mode, as described in [Routing exceptions from EL0 to EL2](#).
 - A debug exception that is explicitly routed to Hyp mode, as described in [Routing debug exceptions to EL2 using AArch32](#).

Note

The virtual exceptions cannot be taken to Hyp mode. They are always taken to an EL1 mode.

D1.11.4.2 The possible modes for taking each exception

Each of the exception descriptions in [AArch32 state exception descriptions](#) includes a subsection that describes the modes to which each exception can be taken. Those subsections are:

- [The PE mode to which the Undefined Instruction exception is taken.](#)
- [The PE mode to which the Hyp Trap exception is taken.](#)
- [The PE mode to which the Supervisor Call exception is taken.](#)
- [The PE mode to which the Hypervisor Call exception is taken.](#)
- [The PE mode to which the Prefetch Abort exception is taken.](#)
- [The PE mode to which the Data Abort exception is taken.](#)
- [The PE mode to which the Virtual SError interrupt exception is taken.](#)
- [The PE mode to which the physical IRQ exception is taken.](#)
- [The PE mode to which the Virtual IRQ exception is taken.](#)
- [The PE mode to which the physical FIQ exception is taken.](#)

- The PE mode to which the Virtual FIQ exception is taken.

These descriptions also show the vector offset for the exception entry for each mode.

For more information about:

- Vector offsets, see [Exception vectors and the exception base address](#).
- The routing of synchronous External aborts or SError, IRQ, and FIQ interrupt exceptions, and the virtual exceptions, see [Asynchronous exception routing controls](#).

D1.11.4.2.1 UNPREDICTABLE cases when the value of HCR.TGE is 1

When the value of HCR.TGE is 1, exceptions that would otherwise be taken to EL1 are, instead, routed to EL2, see [Routing exceptions from EL0 to EL2](#). Related to this, when the value of HCR.TGE is 1, execution in an EL1 mode is UNPREDICTABLE. The architecture does not constrain this UNPREDICTABLE behavior, but software that follows the Arm recommendations cannot get to this state. When following the Arm recommendations, any attempt to move to an EL1 mode when the value of HCR.TGE is 1 is either:

- An illegal exception return, see [Illegal return events from AArch32 state](#).
- An illegal PE mode change, see [Illegal changes to PSTATE.M](#).

D1.11.5 PE state on exception entry

The description of each exception includes a pseudocode description of entry to that exception, as [Table D1-6](#) shows. The following sections describe the PE state changes on entering an exception, for different implementations and operating states. However, you must always see the exception entry pseudocode for a full description of the state changes on exception entry:

- [Instruction set state on exception entry](#).
- [PSTATE.E value on exception entry](#).
- [PSTATE.{A, I, F, M} values on exception entry](#).

D1.11.5.1 Instruction set state on exception entry

Exception handlers can execute in either T32 state or A32 state. On exception entry, PSTATE.T is set to the required value, as determined by SCTLR.TE or HSCTLR.TE, depending on the mode the exception is taken to. [Table D1-9](#) shows this:

Table D1-9 PSTATE.T bit value on exception entry

Mode to which exception is taken	HSCTLR.TE	SCTLR.TE	PSTATE.T	Exception handler state
Not Hyp mode	x	0	0	A32
		1	1	T32
Hyp mode	0	x	0	A32
	1	x	1	T32

For an exception taken to a PE mode other than Hyp mode, the SCTLR.TE bit determines the instruction set state for the exception handler.

D1.11.5.2 PSTATE.E value on exception entry

PSTATE.E controls the load and store endianness for data handling. Table D1-10 show the value to which this bit is set on exception entry:

Table D1-10 PSTATE.E value on exception entry

Exception mode	HSCTLR.EE	SCTLR.EE	Endianness for data loads and stores	PSTATE.E
EL1	x	0	Little-endian	0
		1	Big-endian	1
Hyp	0	x	Little-endian	0
	1	x	Big-endian	1

For more information, see the bit description in [Saved Program Status Registers \(SPSRs\)](#).

D1.11.5.3 PSTATE.{A, I, F, M} values on exception entry

On exception entry, PSTATE.M is set to the value for the mode to which the exception is taken, as described in [PE mode for taking exceptions](#).

Table D1-11 shows the cases where PSTATE.{A, I, F} bits are set to 1 on an exception entry, and how this depends on the mode and Security state to which an exception is taken. If the table entry for a particular mode and Security state does not define a value for a PSTATE.{A, I, F} bit then that bit is unchanged by the exception entry.

In this table, the *PE mode exception is taken to* column is the mode to which the exception is taken.

Table D1-11 PSTATE.{A, I, F} values on exception entry

PE mode exception is taken to	
Hyp	PSTATE.A is set to 1 PSTATE.I is set to 1 PSTATE.F is set to 1
FIQ	PSTATE.A is set to 1 PSTATE.I is set to 1 PSTATE.F is set to 1
IRQ, Abort	PSTATE.A is set to 1 PSTATE.I is set to 1
Undefined, Supervisor	PSTATE.I is set to 1

[Asynchronous exception behavior for exceptions taken from AArch32 state](#) describes how, in some situations, the PSTATE.{A, I, F} bits mask the taking of SError exceptions, IRQ interrupts, and FIQ interrupts.

D1.11.6 Routing exceptions from EL0 to EL2

Note

The routing control described in this section permits a usage model where applications execute in User mode under a hypervisor, which executes in Hyp mode, without a Guest RTOS running at EL1. This control applies when the PE is executing in EL0 and the value of HCR.TGE is 1.

If the PE is in User mode, any exception that would otherwise be taken to EL1 is taken to EL2 if the value of [HCR.TGE](#) is 1.

In this case the exception is taken to Hyp mode, instead of to the default mode for handling the exception. For more information, see [Exception reporting when HCR.TGE routes an exception to EL2 using AArch32](#).

When the value of [HCR.TGE](#) is 1, meaning TGE routing from EL0 using AArch32 to EL2 using AArch32 applies:

- The [SCTLR.M](#) bit is treated as 0 for all purposes other than a direct read of the [SCTLR](#) register.
- Each of the [HCR](#).{FMO, IMO, AMO} bits is treated as 1 for all purposes other than a direct read of the [HCR](#) register.
- Each of the [HDCR](#).{TDE, TDA, TDRA, TDOSA} bits is treated as 1 for all purposes other than a direct read of the [HDCR](#) register.
- An exception return to EL1 is treated as an illegal exception return, see [Illegal return events from AArch32 state](#).
- All virtual interrupts, including any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts, are disabled.

D1.11.6.1 Exception reporting when HCR.TGE routes an exception to EL2 using AArch32

The following sections give more information about the behavior of synchronous exceptions that are routed to Hyp mode because the value of [HCR.TGE](#) is 1:

- [Undefined Instruction exception, when the value of HCR.TGE is 1](#).
- [Supervisor Call exception, when the value of HCR.TGE is 1](#).
- [Abort exceptions, when the value of HCR.TGE is 1](#).
- [Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1](#).

D1.11.6.1.1 Undefined Instruction exception, when the value of HCR.TGE is 1

When [HCR.TGE](#) is set to 1, if the PE is executing in User mode and attempts to execute an UNDEFINED instruction, it takes the Hyp Trap exception, instead of an Undefined Instruction exception. On taking the Hyp Trap exception, the [HSR](#) reports an unknown reason for the exception, using the EC value 0x00. For more information, see [Use of the HSR](#).

D1.11.6.1.2 Supervisor Call exception, when the value of HCR.TGE is 1

When [HCR.TGE](#) is set to 1, if the PE executes an SVC instruction in User mode, the Supervisor Call exception generated by the instruction is taken to Hyp mode.

The [HSR](#) reports that entry to Hyp mode was because of a Supervisor Call exception, and:

- If the SVC is unconditional, takes for the `imm16` value in the [HSR](#):
 - A zero-extended 8-bit immediate value for the T32 SVC instruction.

Note

The only T32 encoding for SVC is a 16-bit instruction encoding.

 - The bottom16 bits of the immediate value for the A32 SVC instruction.
- If the SVC is conditional, the `imm16` value in the [HSR](#) is UNKNOWN.

If the SVC is conditional, the PE takes the exception only if the instruction passes its Condition code check.

The [HSR](#) reports the exception as a Supervisor Call exception taken to Hyp mode, using the EC value 0x11. For more information, see [Use of the HSR](#).

Note

The effect of setting [HCR.TGE](#) to 1 is to route the Supervisor Call exception to Hyp mode, not to trap the execution of the SVC instruction. This means that the preferred return address for the exception, when routed to Hyp mode in this way, is the instruction after the SVC instruction.

D1.11.6.1.3 Abort exceptions, when the value of HCR.TGE is 1

When the value of [HCR.TGE](#) is 1, if the PE is executing in User mode then any abort exception generates an exception that is taken as a Hyp Trap exception. Where an attempt to execute an instruction causes an abort, on taking the Hyp Trap exception, the [HSR](#) indicates whether a Data Abort exception or a Prefetch Abort exception caused the Hyp Trap exception entry, and presents a valid syndrome in the [HSR](#).

An Error exception that is routed to Hyp mode because the value of [HCR.TGE](#) is 1 is reported as a Data Abort exception routed to Hyp mode.

The [HSR](#) reports the exception either:

- As a Prefetch Abort exception routed to Hyp mode, using the EC value `0x20`.
- As a Data Abort exception routed to Hyp mode, using the EC value `0x24`.

For more information about the exception reporting, see [Use of the HSR](#).

D1.11.6.1.4 Reporting of exceptions routed to EL2 using AArch32 because the value of HCR.TGE is 1

[PL1 configurable controls](#) describes controls that, when the value of [HCR.TGE](#) is 0, can generate exceptions that are taken from EL0 to EL1. When EL2 is using AArch32 and the value of [HCR.TGE](#) is 1, the exceptions generated by these controls are routed to Hyp mode. [Table D1-12](#) shows how these exceptions are then reported in the [HSR](#).

Table D1-12 Syndrome reporting in HSR from HCR.TGE routing of traps, disables, and enables

Control provided by PL1	Control type ^a	Syndrome reporting in HSR
SCTLR .{nTWE, nTWI}	T	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
SCTLR .{SED, ITD}	D	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
SCTLR .CP15BEN	E	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
CPACR .TRCDIS	T	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
CPACR .{cp11, cp10}	E	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
FPEXC .EN	E	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
CPACR .ASEDIS	D	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
DBGDSCR ext.UDCCdis	T	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
CNTKCTL .{PL0PTEN, PL0VTEN, PL0PCTEN, PL0VCTEN}	T	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason
PMUSERENR .{ER, CR, SW, EN}	T	Uses EC syndrome value <code>0x00</code> , Exception for an unknown reason

a. T indicates a trap control, E indicates an instruction enable, and D indicates an instruction disable. For the definition of these terms, see the list that begins with [Instruction enables and instruction disables](#).

D1.11.7 Routing debug exceptions to EL2 using AArch32

When the value of [HDCR.TDE](#) is 1, if the PE is executing in a mode other than Hyp mode, any Debug exception is routed to Hyp mode. This means it generates a Hyp Trap exception. This applies to:

- Debug exceptions associated with an instruction fetch, that would otherwise generate a Prefetch Abort exception. These are the Breakpoint, Breakpoint Instruction, and Vector Catch exception, see [AArch32 Self-hosted Debug](#).
- Watchpoint exceptions associated with data accesses, that would otherwise generate a Data Abort exception. See [Watchpoint exceptions](#).

When the value of [HDCR.TDE](#) is 1, each of the [HDCR](#).{TDRA, TDOSA, TDA} bits is treated as 1 for all purposes other than reading the [HDCR](#) register.

Note

- A Breakpoint or Watchpoint debug event that generates entry to Debug state cannot be trapped to Hyp mode. See [Breakpoint and Watchpoint debug events](#).
 - When [HDCR.TDE](#) is set to 1, the Hyp Trap exception is generated instead of the Prefetch Abort exception or Data Abort exception that is otherwise generated by the Debug exception.
 - Debug exceptions, other than Breakpoint Instruction exceptions, are never generated in Hyp mode.
-

When a Hyp Trap exception is generated because the value of [HDCR.TDE](#) is 1, The [HSR](#) reports the exception either:

- As a Prefetch Abort exception routed to Hyp mode, using the EC value `0x20`.
- As a Data Abort exception routed to Hyp mode, using the EC value `0x24`.

For more information, see [Use of the HSR](#).

D1.12 Routing of aborts

A memory abort is either a Data Abort exception or a Prefetch Abort exception. When executing, depending on the cause of the abort, and possibly on configuration settings, an abort is taken either:

- To the Exception level of the PE mode from which the abort is taken. In this case the abort is taken to AArch32 state.
- To a higher Exception level.

The mode to which a memory abort is taken depends on the reason for the exception, the mode the PE is in when it takes the exception, and configuration settings, as follows:

Memory aborts taken to Hyp mode

The following aborts are taken to EL2. This means they are taken to Hyp mode:

- Alignment faults taken:
 - When the PE is in Hyp mode.
 - When the PE is in a PL1 or EL0 mode and the exception is generated because the PL1&0 stage 2 translation identifies the target of an unaligned access as any type of Device memory.
 - When the PE is in User mode and [HCR.TGE](#) is set to 1. For more information, see [Abort exceptions, when the value of HCR.TGE is 1](#).
- When the PE is using the PL1&0 translation regime:
 - [MPU faults](#) from stage 2 translations, for which the stage 1 translation did not cause an MPU fault.
- When the PE is using EL2 translation regime, [MPU faults](#) from stage 1 translations.
- Debug exceptions, if [HDCR.TDE](#) is set to 1. For more information, see [Routing debug exceptions to EL2 using AArch32](#).

Memory aborts taken to Abort mode

All memory aborts that are taken to an Exception level that is using AArch32 are taken to Abort mode. The following aborts are taken to Abort mode:

- When the PE is in a PL1 or EL0 mode, Alignment faults taken for any of the following reasons:
 - [SCTLR.A](#) is set to 1.
 - An instruction that does not support unaligned accesses is committed for execution, and the instruction accesses an unaligned address.
 - The PL1&0 stage 1 translation identifies the target of an unaligned access as any type of Device memory.

These exceptions are taken to Abort mode only if the value of [HCR.TGE](#) is 0.

- When the PE is using the PL1&0 translation regime, an MPU fault from a stage 1 translation.
- External aborts, if the PE was executing in a PL1 or EL0 mode when it took the exception and the abort is not taken to EL2 for one of the reasons defined in [Memory aborts taken to Hyp mode](#).
- Virtual Aborts, see [Virtual exceptions when an implementation includes EL2](#).
- When the value of [HDCR.TDE](#) is 0, Debug exceptions. For more information, see [Routing debug exceptions to EL2 using AArch32](#).

Memory aborts with IMPLEMENTATION DEFINED behavior

In addition, a PE can generate an abort for an IMPLEMENTATION DEFINED reason associated with lockdown. Whether such an abort is taken to Abort mode or is taken to EL2 is IMPLEMENTATION DEFINED, and an implementation might include a mechanism to select whether the abort is routed to Abort mode or to EL2.

When the PE is in a mode other than Hyp mode, if multiple factors cause an Alignment fault, the abort is taken to Abort mode if any of the factors require the abort to be taken to Abort mode. For example, if the [SCTLR.A](#) bit is set to 1, and the access is an unaligned access to an address that the EL2 MPU marks as Device-nGnRnE, then the abort is taken to Abort mode.

For more information, see [Handling exceptions](#).

D1.13 Exception return to an Exception level using AArch32

In the Arm architecture, *exception return* to an Exception level that is using AArch32 requires the simultaneous restoration of the PC and **PSTATE** to values that are consistent with the desired state of execution on returning from the exception. Typically, exception return involves returning to one of:

- The instruction after the instruction boundary at which an asynchronous exception was taken.
- The instruction following an **SVC HVC** instruction, for an exception generated by one of those instructions.
- The instruction that caused the exception, after the reason for the exception has been removed.
- The subsequent instruction, if the instruction that caused the exception has been emulated in the exception handler.

The Arm architecture defines a *preferred return address* for each exception other than Reset, see [Link values saved on exception entry](#). The values of the **SPSR.IT**[7:0] bits generated on exception entry are always correct for this preferred return address, but might require adjustment by the exception handler if returning elsewhere.

In some cases, to calculate the appropriate preferred return address for a return to an Exception level that is using AArch32, a subtraction must be performed on the link value saved on taking the exception. The description of each exception includes any value that must be subtracted from the link value, and other information about the required exception return.

On an exception return, the **PSTATE** takes either:

- The value loaded by the **RFE** instruction.
- If the exception return is not performed by executing an **RFE** instruction, the value of the current **SPSR** at the time of the exception return.

[Illegal return events from AArch32 state](#) describes the behavior if the restored PE state would not be valid for the Exception level, PE mode, and Security state targeted by the exception return.

D1.13.1 Exception return instructions

The instructions that an exception handler can use to return from an exception depend on whether the exception was taken to an EL1 mode, or in an EL2 mode, see:

- [Return from an exception taken to a PE mode other than Hyp mode.](#)
- [Return from an exception taken to Hyp mode.](#)

D1.13.1.1 Return from an exception taken to a PE mode other than Hyp mode

For an exception taken to a PE mode other than Hyp mode, the Arm AArch32 architecture provides the following *exception return instructions*:

- From privileged modes other than System mode, the **ERET** instruction. After the exception return, execution resumes from the address held in the LR (R14) for the mode in which **ERET** is executed. See **ERET**.
- Data-processing instructions with the S bit set and the PC as a destination, see **MOV**, **MOVS (register)** and **SUB**, **SUBS (immediate)**.

Note

The A32 instruction set includes other instructions that can be used for an exception return, but Arm deprecates any use of those instructions.

Typically:

- A return where no subtraction is required uses **SUBS** with an operand of 0, or the equivalent **MOVS** instruction.
- A return requiring subtraction uses **SUBS** with a nonzero operand.
- The **RFE** instruction, see **RFE**, **RFEDA**, **RFEDB**, **RFEIA**, **RFEIB**. If a subtraction is required, typically it is performed before saving the LR value to memory. After the exception return, execution resumes from the address held in the memory location indicated by the base register specified by the **RFE** instruction.
- In A32 state, a form of the **LDM** instruction in which the PC is one of the registers loaded, see **LDM (exception return)**. If a subtraction is required, typically it is performed before saving the LR value to memory.

D1.13.1.2 Return from an exception taken to Hyp mode

For an exception taken to Hyp mode, the Arm architecture provides the `ERET` instruction, see `ERET`. An exception handler executing in Hyp mode must return using the `ERET` instruction.

Hyp mode is implemented only as part of EL2.

D1.13.2 Alignment of exception returns

The `T` bit of the value transferred to the `PSTATE` by an exception return controls the target instruction set of that return. The behavior of the hardware for exception returns for different values of the `T` bit is as follows:

- | | |
|---------------|---|
| T == 0 | The target instruction set state is A32 state. Bits[1:0] of the address transferred to the PC are ignored by the hardware. |
| T == 1 | The target instruction set state is T32 state: <ul style="list-style-type: none"> • Bit[0] of the address transferred to the PC is ignored by the hardware. • Bit[1] of the address transferred to the PC is part of the instruction address. |

Note

In previous versions of the Arm architecture, the `PSTATE.{J, T}` bits determined the Instruction set state. From the introduction of Armv8, `PSTATE.J` is `RES0`.

Arm deprecates any dependence on the requirements that the hardware ignores bits of the address. Arm recommends that the address transferred to the PC for an exception return is correctly aligned for the target instruction set.

After an exception entry other than Reset, the LR value has the correct alignment for the instruction set indicated by the `SPSR.T` bit. This means that if exception return instructions are used with the LR and `SPSR` values produced by such an exception entry, the only precaution software needs to take to ensure correct alignment is that any subtraction is of a multiple of four if returning to A32 state, or a multiple of two if returning to T32 state.

D1.13.3 Illegal return events from AArch32 state

Throughout this section:

- Return** In AArch32 state, refers to any of:
- Execution of any exception return instruction.
 - Execution of a `DRPS` instruction in Debug state.
 - Exit from Debug state.
- If an exception or debug return from an Exception level using AArch32 triggers an illegal exception return, then bit[1] of the PC is either:
- Zero.
 - The value of bit[1] of the return address for the exception or debug return.

The choice between these two alternatives is made by the implementation, and might differ from instance to instance of an illegal exception return.

Note

This means software must support both alternatives.

Saved process state value In AArch32 state, refers to any of:

- The value held in the `SPSR` for any exception return other than an exception return made by executing an `RFE` instruction.
- The value read from memory that is to be restored to `PSTATE` by the execution of an `RFE` instruction.
- The value held in the `SPSR` for the execution of a `DRPS` instruction in Debug state.

- The value held in the [DPSR](#) for a Debug state exit.

Link address In AArch32 state, refers to any of:

- The address held in the link register for any exception return other than an exception return made by executing an `ERET`, `LDM`, or `RFE` instruction.
- The address held in [ELR_hyp](#) for any exception return made by executing an `ERET` instruction.
- The address read from memory that is to be restored to the PC by the execution of an `LDM` or `RFE` instruction.
- The address held in the [DLR](#) for Debug state exit.

Configured from reset

Indicates the state determined on powerup or reset by a configuration input signal, or by another IMPLEMENTATION DEFINED mechanism.

The architecture has a generic mechanism for handling exception or debug returns to a mode or state that is illegal. In AArch32 state, this can occur as a result of any of the following situations:

- A return where the Exception level being returned to is higher than the current Exception level.
- A return to EL1 when the value of the [HCR.TGE](#) bit is 1.
- A return where the value of the saved process state `M[4:0]` field is not a valid AArch32 PE mode for the implementation. [Table D1-3](#) shows the valid `M[4:0]` values for AArch32 PE modes.

In these cases:

- [PSTATE.IL](#) is set to 1, to indicate an illegal return.
- [PSTATE.M](#) is unchanged. This means the PE mode does not change.
- The SS bit is handled in the same way as any other exception or debug return, see Software Step exceptions.
- The following [PSTATE](#) bits are restored from the saved process state value:
 - The N, Z, C, V Condition flags.
 - The Q Overflow or saturation flag.
 - The GE Greater than or Equal flags.
 - The E Endianness mapping bit.
 - The A, I, F exception mask bits.
- The [PSTATE.{IT, T}](#) bits are each either:
 - Set to 0.
 - Copied from the saved process state in the `SPSR` for the PE mode in which the exception is handled.

The choice between these two options is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.
- The PC is restored from the link address, unless the illegal return is the execution of a `DRPS` instruction in Debug state.

When the value of the [PSTATE.IL](#) bit is 1, any attempt to execute any instruction results in an Illegal Execution state exception. See [The Illegal Execution state exception](#).

All aspects of the illegal return, other than the effects described in this section, are the same as for a legal return.

D1.13.4 Legal returns that set [PSTATE.IL](#) to 1

In this section, return, saved process state value, and link address have the meaning that is defined in [Illegal return events from AArch32 state](#).

If the IL bit in the saved process state value is 1, then it is copied to [PSTATE](#) meaning that [PSTATE.IL](#) is set to 1. In this case, the [PSTATE.{IT, T}](#) bits are each either:

- Set to 0.

- Copied from the SPSR, or loaded from memory if the exception return was performed by executing an RFE instruction.

The choice between these two options is determined by an implementation, and might vary dynamically within the implementation. This means software must regard each value as being an UNKNOWN choice between the two permitted values.

Because the return sets the [PSTATE.IL](#) bit to 1, any attempt to execute any instruction results in an Illegal Execution state exception. See [The Illegal Execution state exception](#).

D1.13.5 The Illegal Execution state exception

When the value of the [PSTATE.IL](#) bit is 1, any attempt to execute an instruction generates an Illegal Execution state exception. In AArch32 state, the [PSTATE.IL](#) bit can be set to 1 by one of the following:

- An illegal return, as described in [Illegal return events from AArch32 state](#).
- An illegal change to [PSTATE.M](#), as described in [Illegal changes to PSTATE.M](#).
- A legal return that sets [PSTATE.IL](#) to 1, as described in [Legal returns that set PSTATE.IL to 1](#).

An Illegal Execution state exception is taken in the same way as an Undefined Instruction exception in the current Exception level. If the current Exception level is EL2 using AArch32 state, the [HSR](#) provides additional syndrome information for the exception, see [Use of the HSR](#).

An Illegal Execution state exception has priority over any other Undefined Instruction exception that might arise from instruction execution.

Note

This section only describes the handling of an Illegal Execution state exception that is taken to an Exception level that is using AArch32 state.

On taking any exception to an Exception level that is using AArch32 state:

1. The value of the [PSTATE.IL](#) bit is 1 and this is copied to the SPSR.IL bit for the PE mode to which the exception is taken.
2. The [PSTATE.IL](#) bit is cleared to 0.

Note

This means that it is not possible for software to observe the value of [PSTATE.IL](#).

D1.13.5.1 Pseudocode description of exception return

The [AArch32_ExceptionReturn\(\)](#) function transfers the return address to the PC and restores [PSTATE](#) to its saved value.

This function uses the function [SetPSTATEFromPSR\(\)](#).

The [IllegalExceptionReturn\(\)](#) function checks for an Illegal Execution state exception.

[R-profile Architecture Pseudocode](#) includes the definitions of these functions.

D1.14 Asynchronous exception behavior for exceptions taken from AArch32 state

[PSTATE](#).{A, I, F} bits *mask* the asynchronous exceptions, meaning that when the value of one of these [PSTATE](#) bits is 1, the corresponding exception is not taken.

If a masked asynchronous exception remains signaled, then the exception remains pending unless the value of the [PSTATE](#) bit is changed to 0.

EL2 provides controls that affect:

- The routing of these exceptions, see [Asynchronous exception routing controls](#).
- Masking of these exceptions, see [Asynchronous exception masking controls](#).

The register control bits are provided in [HCR](#).

D1.14.1 Virtual exceptions when an implementation includes EL2

When implemented, EL2 provides the following virtual exceptions, which correspond to the physical asynchronous exceptions:

- Virtual SError, which corresponds to a physical external SError exception.
- Virtual IRQ, which corresponds to a physical IRQ.
- Virtual FIQ, which corresponds to a physical FIQ.

When the value of [HCR](#).TGE is 0 and the value of an [HCR](#).{AMO, IMO, FMO} routing control bit is 1, the corresponding virtual interrupt is enabled and a virtual exception is generated either:

- By setting the corresponding virtual interrupt pending bit, [HCR](#).{VA, VI, VF}, to 1.
- For a Virtual IRQ or Virtual FIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller. See, for example, the *ARM Generic Interrupt Controller Architecture Specification*.

When the value of [HCR](#)._EL2.TGE is 1, all virtual interrupts are disabled.

When a virtual interrupt is disabled:

- It cannot be taken.
- It cannot be seen in the ISR.

In AArch32 state, a virtual exception is taken only from a EL1 or EL0 mode. In any other mode, if the exception is generated it is not taken.

A virtual exception is taken to the default mode for the corresponding physical exception. This means:

- A Virtual SError is taken to Abort mode.
- A Virtual IRQ is taken to IRQ mode.
- A Virtual FIQ is taken to FIQ mode.

[Table D1-13](#) summarizes the [HCR](#) bits that route asynchronous exceptions to EL2, and the bits that generate the virtual exceptions.

Table D1-13 HCR bits controlling asynchronous exceptions

Exception	Routing the physical exception to EL2	Generating the virtual exception
SError	HCR .AMO	HCR .VA

Exception	Routing the physical exception to EL2	Generating the virtual exception
IRQ	HCR.IMO	HCR.VI
FIQ	HCR.FMO	HCR.VF

The [HCR.{VA, VI, VF}](#) bits generate a virtual exception only if set to 1 when the value of the corresponding [HCR.{AMO, IMO, FMO}](#) is 1.

When the value of an [HCR.{AMO, IMO, FMO}](#) control bit is 1, the corresponding mask bit in [PSTATE](#):

- Does not mask the physical exception.
- Masks the virtual exception when the PE is executing in an EL1 or EL0 mode.

Taking a Virtual Abort exception clears [HCR.VA](#) to zero. Taking a Virtual IRQ exception or a Virtual FIQ exception does not affect the value of [HCR.VI](#) or [HCR.VF](#).

Note

This means that the exception handler for a Virtual IRQ exception or a Virtual FIQ exception must cause software that is executing at EL2 to update the [HCR](#) to clear the appropriate virtual exception bit to 0.

See [WFE wakeup events](#) and [Wait For Interrupt](#) for information about how virtual exceptions affect wake up from power-saving states.

Note

A hypervisor can use virtual exceptions to signal exceptions to the current Guest RTOS. The Guest RTOS takes a virtual exception exactly as it would take the corresponding physical exception, and is unaware of any distinction between virtual exception and the corresponding physical exception.

D1.14.1.1 Effects of the [HCR.{AMO, IMO, FMO}](#) bits

As described in this section, the [HCR.{AMO, IMO, FMO}](#) bits are part of the mechanism for enabling the virtual exceptions. In addition:

- As mentioned in this section, affect the routing of the exceptions. See [Asynchronous exception routing controls](#).
- Affect the masking of the exceptions. See [Asynchronous exception masking controls](#).

D1.14.2 Asynchronous exception routing controls

The following bits in the [HCR](#) route asynchronous exceptions to EL2 that are taken from an EL1 or EL0 mode.

- [HCR.AMO](#)** When the value of this bit is 1, an SError exception taken from an EL1 or EL0 mode is taken to EL2, instead of to Abort mode.
- [HCR.FMO](#)** When the value of this bit is 1, an FIQ exception taken from an EL1 or EL0 mode is taken to EL2, instead of to FIQ mode.
- [HCR.IMO](#)** When the value of this bit is 1, an IRQ exception taken from an EL1 or EL0 mode is taken to EL2, instead of to IRQ mode.

When the value of one of the [HCR.{AMO, FMO, IMO}](#) bits is 1, the exception is taken to Hyp mode.

Only software executing in Hyp mode can change the values of these bits.

The [HCR.{AMO, FMO, IMO}](#) bits also affect the masking of asynchronous exceptions, as described in [Asynchronous exception masking controls](#).

The [HCR.{AMO, FMO, IMO}](#) bits have no effect on the routing of Virtual Abort, Virtual FIQ, and Virtual IRQ exceptions.

Note

When the PE is in Hyp mode:

- Physical asynchronous exceptions are taken to Hyp mode.
 - Virtual exceptions are not signaled to the PE.
-

See also [Asynchronous exception behavior for exceptions taken from AArch32 state](#).

D1.14.3 Asynchronous exception masking controls

The [PSTATE](#).{A, I, F} bits can mask the taking of the corresponding exceptions from AArch32 state, as follows:

- [PSTATE](#).A can mask SError exceptions.
- [PSTATE](#).I can mask IRQ exceptions.
- [PSTATE](#).F can mask FIQ exceptions.

[HCR](#).{AMO, IMO, FMO} bits modify the masking of exceptions.

D1.14.3.1 Asynchronous exception masking in an implementation that includes EL2

The [HCR](#).{AMO, IMO, FMO} bits modify the effect of the [PSTATE](#).{A, I, F} bits. When the value of an [HCR](#).{AMO, IMO, FMO} mask override bit is 1, the value of the corresponding [PSTATE](#).{A, I, F} bit is ignored when the exception is taken from a mode other than Hyp mode.

D1.15 AArch32 state exception descriptions

[Handling exceptions that are taken to an Exception level using AArch32](#) gives general information about exception handling. This section describes each of the exceptions, in the following subsections:

- [Undefined Instruction exception.](#)
- [Hyp Trap exception.](#)
- [Supervisor Call \(SVC\) exception.](#)
- [Hypervisor Call \(HVC\) exception.](#)
- [Prefetch Abort exception.](#)
- [Data Abort exception.](#)
- [Virtual SError interrupt exception.](#)
- [IRQ exception.](#)
- [Virtual IRQ exception.](#)
- [FIQ exception.](#)
- [Virtual FIQ exception.](#)

[Additional pseudocode functions for exception handling](#) gives additional pseudocode that is used in the pseudocode descriptions of a number of the exceptions.

D1.15.1 Undefined Instruction exception

An Undefined Instruction exception might be caused by:

- A System register access, floating-point, or Advanced SIMD instruction that is not accessible because of the settings in one or more of the [CPACR](#), [NSACR](#), [HCPTR](#), and [DBGDSCRExt](#).
- A System register access, floating-point, or Advanced SIMD instruction that is not implemented.
- A System register access, floating-point, or Advanced SIMD instruction that causes an exception during execution. This includes:
 - Trapped floating-point exceptions that are taken to AArch32, if an implementation supports these traps. See [Floating-point exceptions and exception traps](#).
 - Execution of certain floating-point instructions when one or both of the [FPSCR](#).{Stride, Len} fields in nonzero, in an implementation in which those fields are RW. The description of [FPEXC](#) specifies the instructions to which this applies.
- An instruction that is UNDEFINED.

Note

The Undefined Instruction exception is taken using offset 0x04 in the Hyp vector table. See [The vector tables and exception offsets](#).

By default, an Undefined Instruction exception is taken to Undefined mode, but an Undefined Instruction exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see [The PE mode to which the Undefined Instruction exception is taken](#).

The Undefined Instruction exception can provide:

- Signaling of an illegal instruction execution.
- *Lazy context switching* of System registers.

The preferred return address for an Undefined Instruction exception is the address of the instruction that generated the exception. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from Undefined mode, the exception return uses the [SPSR](#) and LR_und values generated by the exception entry, as follows:

- If `SPSR.T` is 0, indicating that the exception occurred in A32 state, the return uses an exception return instruction with a subtraction of 4.
- If `SPSR.T` is 1, indicating that the exception occurred in T32 state, the return uses an exception return instruction with a subtraction of 2.
- If returning from Hyp mode, the exception return is performed by an `ERET` instruction, using the `SPSR` and `ELR_hyp` values generated by the exception entry.

For more information, see [Exception return to an Exception level using AArch32](#).

Note

If handling the Undefined Instruction exception requires instruction emulation, followed by return to the next instruction after the instruction that caused the exception, the instruction emulator must use the instruction length to calculate the correct return address, and to calculate the updated values of the IT bits if necessary.

D1.15.1.1 The PE mode to which the Undefined Instruction exception is taken

[Figure D1-2](#) shows how the implementation, state, and configuration options determine the PE mode to which an Undefined Instruction exception is taken, when the exception is taken to an Exception level that is using AArch32.

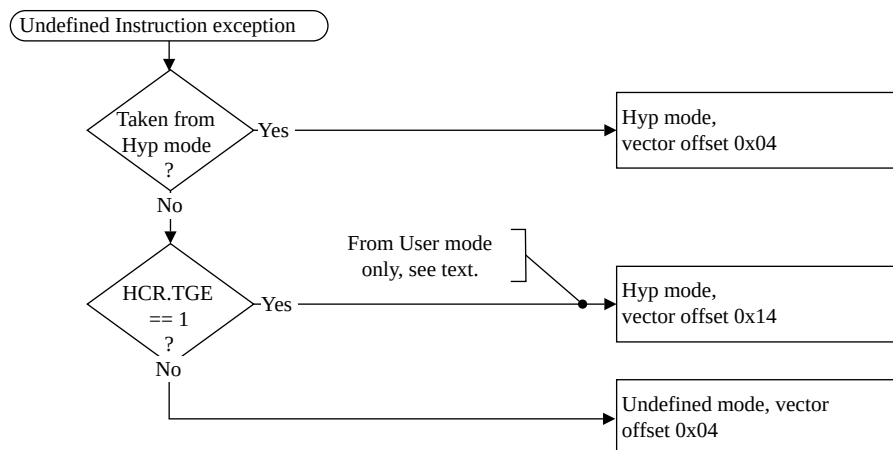


Figure D1-2 The PE mode an Undefined Instruction exception is taken to in AArch32 state

See also [UNPREDICTABLE cases when the value of HCR.TGE is 1](#).

D1.15.1.2 Pseudocode description of taking the Undefined Instruction exception

The `AArch32_Undefined()` pseudocode procedure determines whether the Undefined Instruction exception is taken to AArch32 state. If it is taken to AArch32 state, the `AArch32_TakeUndefInstrException()` pseudocode procedure describes how the PE takes the exception.

D1.15.1.3 Conditional execution of undefined instructions

The conditional execution rules described in [Conditional execution](#) apply to all instructions. This includes undefined instructions and other instructions that would cause entry to the Undefined Instruction exception.

If such an instruction fails its condition check, the behavior depends on the potential cause of entry to the Undefined Instruction exception, as follows:

- If the potential cause is the execution of the instruction itself and depends on data values used by the instruction, the instruction executes as a `NOP` and does not cause an Undefined Instruction exception.
- In the following cases, it is IMPLEMENTATION DEFINED whether the instruction executes as a `NOP` or causes an Undefined Instruction exception:
 - The potential cause is the execution of an earlier System register access instruction, floating-point instruction, or Advanced SIMD instruction.

- The potential cause is the execution of the instruction itself without dependence on the data values used by the instruction.

An implementation must handle all such cases in the same way.

D1.15.1.4 Interaction of UNDEFINED instruction behavior with UNPREDICTABLE or CONSTRAINED UNPREDICTABLE instruction behavior

If this manual describes an instruction as both:

- UNPREDICTABLE and UNDEFINED then the instruction is UNPREDICTABLE.
- CONSTRAINED UNPREDICTABLE and UNDEFINED then the instruction is CONSTRAINED UNPREDICTABLE.

Note

An example of this is where both:

- An instruction, or instruction class, is made UNDEFINED by some general principle, or by a configuration field.
 - A particular encoding of that instruction or instruction class is specified as CONSTRAINED UNPREDICTABLE.
-

D1.15.2 Hyp Trap exception

The Hyp Trap exception provides the standard mechanism for trapping Guest OS functions to the hypervisor.

The Hyp Trap exception is implemented only as part of EL2 and can be generated only if EL2 is using AArch32.

A Hyp Trap exception is generated if the PE is running in a mode other than Hyp mode, and commits for execution an instruction that is trapped to Hyp mode. Instruction traps are enabled by setting bits to 1 in the [HCR](#), [HCPTR](#), [HDCR](#), or [HSTR](#). For more information, see [EL2 configurable controls](#).

The preferred return address for a Hyp Trap exception is the address of the trapped instruction. The exception return is performed by an ERET instruction, using the [SPSR](#) and [ELR_hyp](#) values generated by the exception entry.

Note

The [SPSR](#) and [ELR_hyp](#) values generated on exception entry can be used, without modification, for an exception return to re-execute the trapped instruction. If the exception handler emulates the trapped instruction, and must return to the following instruction, the emulation of the instruction must include modifying [ELR_hyp](#), and possibly updating [SPSR_hyp](#).

When the PE enters the handler for a Hyp Trap exception, the [HSR](#) holds syndrome information for the exception. For more information, see [Use of the HSR](#).

D1.15.2.1 The PE mode to which the Hyp Trap exception is taken

A Hyp Trap exception is taken to Hyp mode, using a vector offset of 0×14 from the Hyp exception base address.

D1.15.2.2 Pseudocode description of taking the Hyp Trap exception

The [AArch32_TakeHypTrapException\(\)](#) pseudocode procedure describes how the PE takes the exception.

D1.15.3 Supervisor Call (SVC) exception

The Supervisor Call instruction, [SVC](#), requests a supervisor function, typically to request an operating system function. When EL1 is using AArch32, executing an [SVC](#) instruction causes the PE to enter Supervisor mode. For more information, see [SVC](#).

Note

In an implementation that includes EL2, when EL2 is using AArch32:

- When an `SVC` instruction is executed in Hyp mode, the Supervisor Call exception is taken to Hyp mode. For more information, see `SVC`.
- When the `HCR.TGE` bit is set to 1, the Supervisor Call exception generated by execution of an `SVC` instruction in User mode is routed to Hyp mode. For more information, see [Supervisor Call exception, when the value of `HCR.TGE` is 1](#).

By default, a Supervisor Call exception that is taken to AArch32 state is taken to Supervisor mode, but a Supervisor Call exception can be taken to EL2, meaning it is taken to Hyp mode if EL2 is using AArch32, see [The PE mode to which the Supervisor Call exception is taken](#).

The preferred return address for a Supervisor Call exception is the address of the next instruction after the `SVC` instruction. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from Supervisor mode, the exception return uses the `SPSR` and `LR_svc` values generated by the exception entry, in an exception return instruction without subtraction.
- If returning from Hyp mode, the exception return is performed by an `ERET` instruction, using the `SPSR` and `ELR_hyp` values generated by the exception entry.

For more information, see [Exception return to an Exception level using AArch32](#).

D1.15.3.1 The PE mode to which the Supervisor Call exception is taken

[Figure D1-3](#) shows how the implementation, state, and configuration options determine the PE mode to which a Supervisor Call exception is taken, when the exception is taken to an Exception level that is using AArch32.

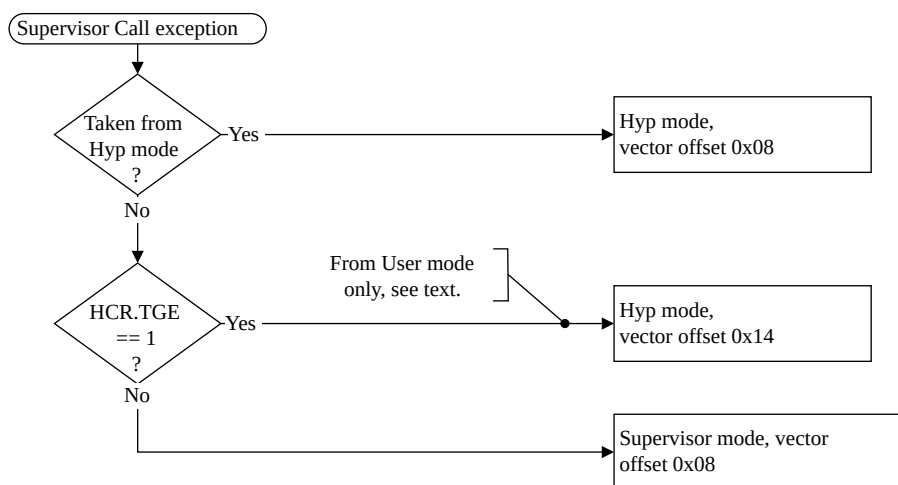


Figure D1-3 The PE mode the Supervisor Call exception is taken to in AArch32 state

See also [UNPREDICTABLE cases when the value of `HCR.TGE` is 1](#).

D1.15.3.2 Pseudocode description of taking the Supervisor Call exception

The `AArch32_CallSupervisor()` pseudocode procedure determines whether the Supervisor Call exception is taken to AArch32 state. If it is taken to AArch32 state, the `AArch32_TakeSVCException()` pseudocode procedure describes how the PE takes the exception.

D1.15.4 Hypervisor Call (HVC) exception

The Hypervisor Call exception is implemented only as part of EL2.

The Hypervisor Call instruction, `HVC`, requests a hypervisor function. When EL2 is using AArch32, executing an `HVC` instruction generates a Hypervisor Call exception that is taken to Hyp mode. For more information, see [HVC](#).

The preferred return address for a Hypervisor Call exception is the address of the next instruction after the `HVC` instruction. The exception return is performed by an `ERET` instruction, using the `SPSR` and `ELR_hyp` values generated by the exception entry.

For more information, see [Exception return to an Exception level using AArch32](#).

When EL2 is using AArch32, executing an `HVC` instruction transfers the immediate argument of the instruction to the `HSR`. The exception handler retrieves the argument from the `HSR`, and therefore does not have to access the original `HVC` instruction. For more information, see [Use of the HSR](#).

D1.15.4.1 The PE mode to which the Hypervisor Call exception is taken

The Hypervisor Call exception is supported only as part of EL2. When EL2 is using AArch32, a Hypervisor Call exception is taken to Hyp mode, using a vector offset that depends on the mode from which the exception is taken, as [Figure D1-4](#) shows. This offset is from the Hyp exception base address.

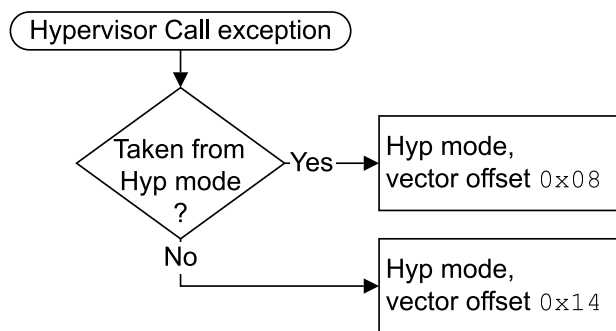


Figure D1-4 The PE mode the Hypervisor Call exception is taken to in AArch32 state

D1.15.4.2 Pseudocode description of taking the Hypervisor Call exception

The `AArch32_TakeHVCEXception()` pseudocode procedure describes how the PE takes a Hypervisor Call exception.

D1.15.5 Prefetch Abort exception

A Prefetch Abort exception can be generated by:

- A synchronous memory abort on an instruction fetch.

Note

Asynchronous External aborts on instruction fetches are reported as SError exceptions using the Data Abort exception, see [Data Abort exception](#).

A Prefetch Abort exception entry is synchronous to the instruction whose fetch aborted.

For more information about memory aborts see [MPU faults](#).

- A Breakpoint, Vector Catch or Breakpoint Instruction exception, see [AArch32 Self-hosted Debug](#).

Note

If an implementation fetches instructions speculatively, it must handle a synchronous abort on such an instruction fetch by:

- Generating a Prefetch Abort exception only if the instruction would be executed in a simple sequential execution of the program.
 - Ignoring the abort if the instruction would not be executed in a simple sequential execution of the program.
-

By default a Prefetch Abort exception is taken to Abort mode, but a Prefetch Abort exception can be taken to EL2, meaning it is taken to Hyp mode.

For more information:

- About cases where the Prefetch Abort exception is taken to an Exception level that is using AArch32, see [The PE mode to which the Prefetch Abort exception is taken](#).

The preferred return address for a Prefetch Abort exception is the address of the aborted instruction. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the [SPSR](#) and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
 - [SPSR_abt](#) and [LR_abt](#) if returning from Abort mode.
- If returning from Hyp mode, using the [SPSR_hyp](#) and [ELR_hyp](#) values generated by the exception entry, using an [ERET](#) instruction.

For more information about the handling of Prefetch Abort exceptions in AArch32 state, see [Exception return to an Exception level using AArch32](#).

D1.15.5.1 Prefetch Abort exception reporting a PC alignment fault exception

A PC alignment fault exception that is taken to an Exception level that is using AArch32 is reported as a Prefetch Abort exception, and:

If the exception is taken to EL1

- The IFSR indicates the cause of the exception and [IFSR.STATUS](#) takes the value 0b100001.
- [IFAR](#) holds the value of the address that faulted, including the misaligned low order bit or bits.
- [R14_abt](#) holds the address that faulted, including the misaligned low order bit or bits, with the standard offset for a Prefetch Abort exception.

If the exception is taken to EL2

- [HSR.EC](#) takes the value 0b100010.
- [HSR.IL](#) is UNKNOWN.
- [HSR.ISS](#) is RES0.
- [HIFAR](#) and [ELR_hyp](#) each hold the value of the address that faulted, including the misaligned low order bit or bits.

For a PC alignment fault exception taken to an Exception level that is using AArch32:

- If the exception occurred because of the [CONSTRAINED UNPREDICTABLE](#) behavior of a branch to an unaligned PC value, as described in [Branching to an unaligned PC](#), then bit[0] of the faulting address is forced to zero, and therefore the misalignment is because the value of bit[1] of this address is 1.
- If the exception occurred on an exit from Debug state, as described in [Exiting Debug state](#), then it is [CONSTRAINED UNPREDICTABLE](#) whether bit[0] of the faulting address is forced to zero.

D1.15.5.2 The PE mode to which the Prefetch Abort exception is taken

Figure D1-5 shows how the implementation, state, and configuration options determine the PE mode to which a Prefetch Abort exception is taken, when the exception is taken to an Exception level that is using AArch32.

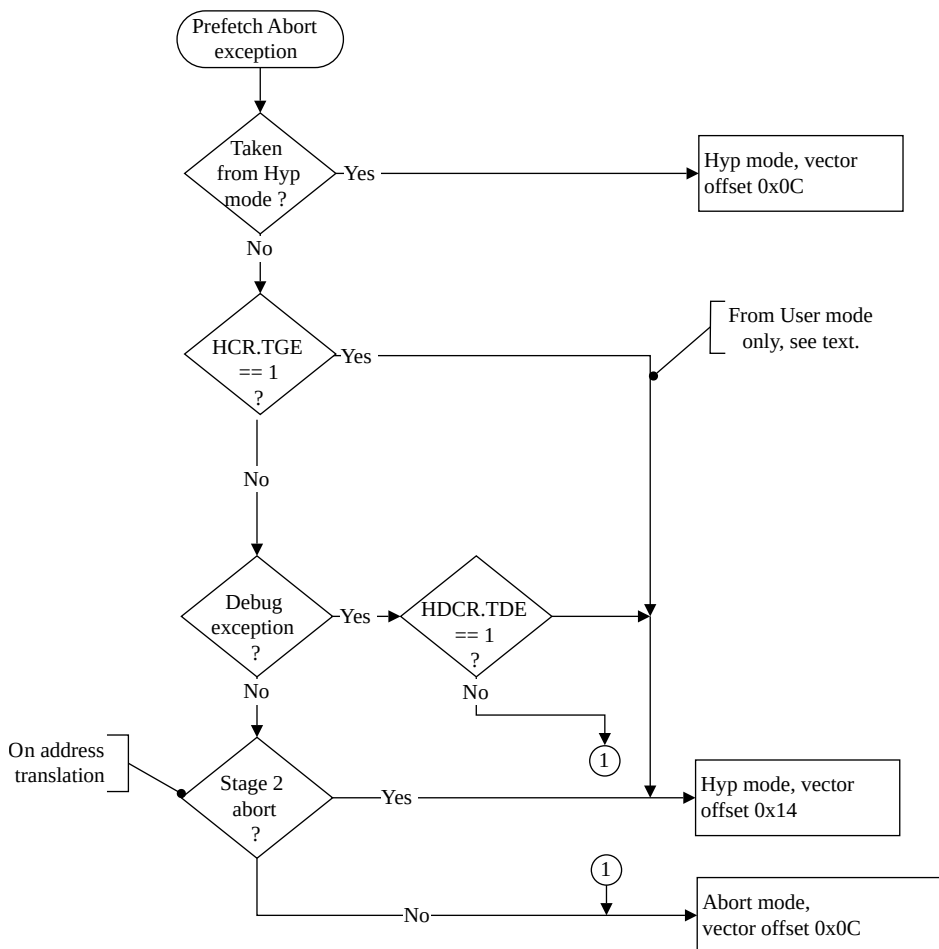


Figure D1-5 The PE mode the Prefetch Abort exception is taken to in AArch32 state

See also [UNPREDICTABLE cases when the value of HCR.TGE is 1](#).

D1.15.5.3 Pseudocode description of taking the Prefetch Abort exception

When the exception is taken in AArch32 state, the `AArch32_TakePrefetchAbortException()` pseudocode procedure describes how the PE takes the exception.

D1.15.6 Data Abort exception

In AArch32 state, a Data Abort exception can be generated by:

- A synchronous abort on a data read or write memory access. Exception entry is synchronous to the instruction that generated the memory access.
- An SError exception. The SError exception might be caused by an External abort on a memory access, which can be any of:
 - A data read or write access.
 - An instruction fetch.
 - In a PMSA memory system, an MPU translation.

Exception entry occurs asynchronously.

As described in [Asynchronous exception masking controls](#), SError exceptions can be masked. When this happens, a generated SError exception is not taken until it is not masked.

- A watchpoint, see [Watchpoint exceptions](#).

By default, when EL1 is using AArch32 a Data Abort exception is taken to Abort mode, but a Data Abort exception can be taken to EL2, meaning it is taken to Hyp mode.

For more information:

- About cases where the Data Abort exception is taken to an Exception level that is using AArch32, see [The PE mode to which the Data Abort exception is taken](#).
- About memory aborts in AArch32 state, see [External abort on a translation](#).

The preferred return address for a Data Abort exception is the address of the instruction that generated the aborting memory access, or the address of the instruction following the instruction boundary at which an SError exception was taken. For an exception taken to AArch32 state, this return is performed as follows:

- If returning from a mode other than Hyp mode, using the [SPSR](#) and LR values generated by the exception entry, using an exception return instruction with a subtraction of 8. This means using:
 - [SPSR_abt](#) and [LR_abt](#) if returning from Abort mode.
- If returning from Hyp mode, using the [SPSR_hyp](#) and [ELR_hyp](#) values generated by the exception entry, using an `ERET` instruction.

For more information about the handling of Data Abort exceptions in AArch32 state, see [Exception return to an Exception level using AArch32](#).

D1.15.6.1 The PE mode to which the Data Abort exception is taken

[Figure D1-6](#) shows the determination of the mode to which a Data Abort exception is taken when the exception is taken to an Exception level that is using AArch32.

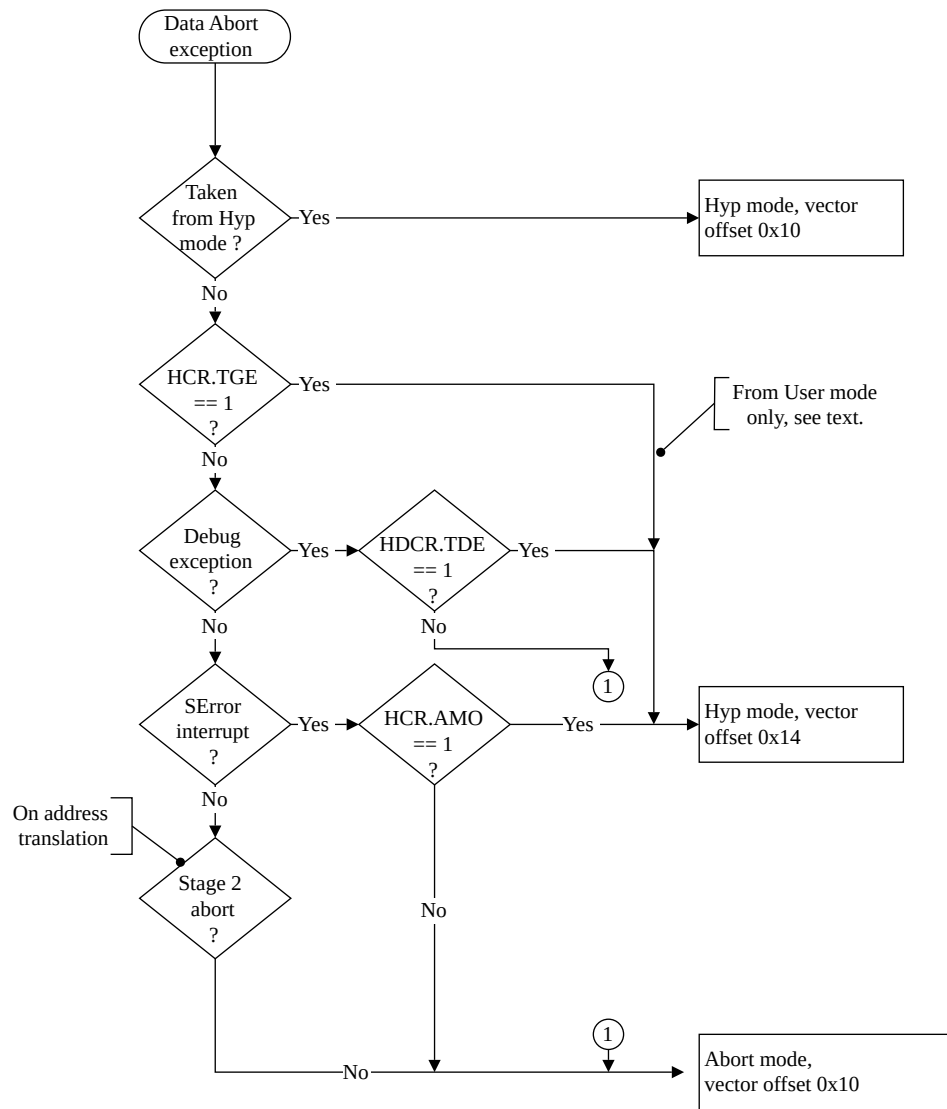


Figure D1-6 The PE mode the Data Abort exception is taken to in AArch32 state

See also [UNPREDICTABLE cases when the value of HCR.TGE is 1](#).

D1.15.6.2 Pseudocode description of taking the Data Abort exception

When the exception is taken in AArch32 state, the [AArch32_TakeDataAbortException\(\)](#) pseudocode procedure describes how the PE takes the exception.

D1.15.6.3 Effects of data-aborted instructions

An instruction that accesses data memory can modify memory by storing one or more values. If the execution of such an instruction generates a Data Abort exception, or causes Debug state entry because of a watchpoint set on the location, the value of each memory location that the instruction stores to is:

- Unchanged for any location for which one of the following applies:
 - An Alignment fault is generated.
 - An [MPU fault](#) is generated.
 - A Watchpoint is generated.
 - An External abort is generated, if that External abort is taken synchronously.
- UNKNOWN for any location for which no exception and no debug event is generated.

If the access to a memory location generates an External abort that is taken asynchronously, it is outside the scope of the architecture to define the effect of the store on that memory location, because this depends on the system-specific nature of the External abort. However, in general, Arm recommends that such locations are unchanged.

For External aborts and Watchpoints, where in principle faulting could be identified at byte or halfword granularity, the size of a location in this definition is the size for which a memory access is single-copy atomic.

In AArch32 state, instructions that access data memory can modify registers in the following ways:

- By loading values into one or more of the general-purpose registers. The registers loaded can include the PC.
- By loading values into one or more of the registers in the Advanced SIMD and floating-point register file.
- By specifying *base register writeback*, in which the base register used in the address calculation has a modified value written to it. All instructions that support base register writeback have **CONSTRAINED UNPREDICTABLE** results if base register writeback is specified with the PC as the base register. Only general-purpose registers can be modified reliably in this way.
- By a direct transfer to or from the Debug Communication Channel (DCC) register, using the LDC and STC instructions. For more information, see [The Debug Communication Channel and Instruction Transfer Register](#). If the instruction that accesses the DCC System registers is an LDC or STC instruction, **UNKNOWN** values are left in the Data Transfer Register and DCC flow-control flags.
- By modifying **PSTATE**.

If the execution of such an instruction generates a synchronous Data Abort exception, the following rules determine the values left in these registers:

- On entry to the Data Abort exception handler:
 - The PC value is the Data Abort vector address, see [Exception vectors and the exception base address](#).
 - The LR_abt value is determined from the address of the aborted instruction.
 Neither value is affected by the results of any load specified by the instruction.
- The base register is restored to its original value if either:
 - The aborted instruction is a load and the list of registers to be loaded includes the base register.
 - The base register is being written back.
- If the instruction only loads one general-purpose register the value in that register is unchanged.
- If the instruction loads more than one general-purpose register, **UNKNOWN** values are left in destination registers other than the PC and the base register of the instruction.
- If the instruction affects any registers in the Advanced SIMD and floating-point register file, **UNKNOWN** values are left in the registers that are affected.
- **PSTATE** bits that are not defined as updated on exception entry retain their current value.
- If the instruction is a STREX, STREXB, STREXH, or STREXD, <Rd> is not updated.

After taking a Data Abort exception, the state of the Exclusives monitors is **UNKNOWN**. Therefore, Arm strongly recommends that the abort handler performs a CLREX instruction, or a dummy STREX instruction, to clear the Exclusives monitor state.

D1.15.6.4 The Arm abort model

The abort model used by an Arm PE is described as a *Base Restored Abort Model*. This means that if a synchronous Data Abort exception is generated by executing an instruction that specifies base register writeback, the value in the base register is unchanged.

The abort model applies uniformly across all instructions.

D1.15.7 Virtual SError interrupt exception

The Virtual SError interrupt exception is implemented only as part of EL2 is enabled in the current Security state.

A Virtual SError interrupt exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.

- The value of `PSTATE.A` is 0.
- EL2 is using AArch32 and the values of the `HCR`.{TGE, AMO, VA} bits are {0, 1, 1}.

The preferred return address for a Virtual SError interrupt exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state, this return is performed using the `SPSR` and `LR_abt` values generated by the exception entry, using an exception return instruction without subtraction.

D1.15.7.1 The PE mode to which the Virtual SError interrupt exception is taken

The Virtual SError interrupt exception is taken using a vector offset of `0x10` from the exception base address.

The conditions for generating a Virtual SError interrupt exception in AArch32 state mean the exception is:

- Taken from a EL1 or EL0 mode.
- Taken to Abort mode .

For more information, see [Virtual exceptions when an implementation includes EL2](#).

Note

Because a Virtual SError interrupt exception taken to AArch32 state is always taken to Abort mode, on exception entry the preferred return address is always saved to `LR_abt`.

D1.15.7.2 Pseudocode description of taking the Virtual SError interrupt exception

The `AArch32_TakeVirtualSErrorException()` pseudocode procedure describes how the PE takes the exception.

D1.15.8 IRQ exception

The IRQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an IRQ interrupt request input to the PE.

When an IRQ exception is taken, exception entry is precise to an instruction boundary.

As described in [Asynchronous exception masking controls](#), IRQ exceptions can be masked. When this happens, a generated IRQ exception is not taken until it is not masked.

By default, an IRQ exception is taken to IRQ mode, but an IRQ exception can be taken to Hyp mode.

For more information, see [The PE mode to which the physical IRQ exception is taken](#)

The preferred return address for an IRQ exception is the address of the instruction following the instruction boundary at which the exception was taken. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the `SPSR` and `LR` values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
 - `SPSR_irq` and `LR_irq` if returning from IRQ mode.
- If returning from Hyp mode, using the `SPSR_hyp` and `ELR_hyp` values generated by the exception entry, using an `ERET` instruction.

For more information, see [Exception return to an Exception level using AArch32](#).

D1.15.8.1 The PE mode to which the physical IRQ exception is taken

[Figure D1-7](#) shows how the implementation, state, and configuration options determine the mode to which an IRQ exception is taken when the exception is taken to an Exception level that is using AArch32.

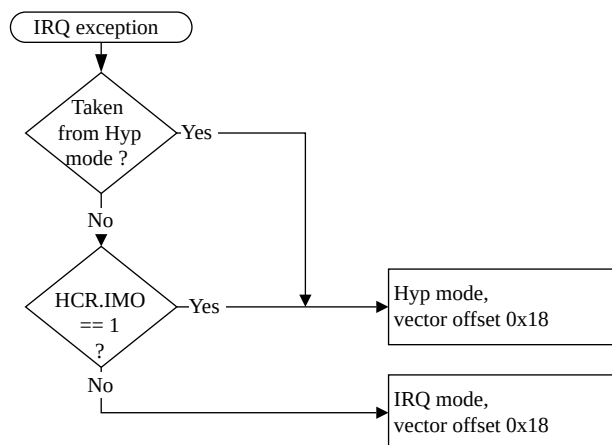


Figure D1-7 The PE mode the IRQ exception is taken to in AArch32 state

D1.15.8.2 Pseudocode description of taking the physical IRQ exception

The `AArch32_TakePhysicalIRQException()` pseudocode procedure describes how the PE takes the exception.

D1.15.9 Virtual IRQ exception

The Virtual IRQ exception is implemented only as part of EL2, if EL2 is enabled in the current Security state.

A Virtual IRQ exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of `PSTATE.I` is 0.
- The value of `HCR.{TGE, IMO}` is {0, 1}.
- Either:
 - `HCR.VI` is 1.
 - A Virtual IRQ exception is generated by an IMPLEMENTATION DEFINED mechanism.

The preferred return address for a Virtual IRQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state this return is performed using the `SPSR` and `LR_irq` values generated by the exception entry, using an exception return instruction with a subtraction of 4.

D1.15.9.1 The PE mode to which the Virtual IRQ exception is taken

The Virtual IRQ exception uses a vector offset of `0x18`.

The conditions for generating a Virtual IRQ exception in AArch32 state mean the exception is:

- Taken from an EL1 or EL0 mode.
- Taken to IRQ mode.

For more information, see [Virtual exceptions when an implementation includes EL2](#).

D1.15.9.2 Pseudocode description of taking the Virtual IRQ exception

The `AArch32_TakeVirtualIRQException()` pseudocode procedure describes how the PE takes the exception.

D1.15.10 FIQ exception

The FIQ exception is generated by IMPLEMENTATION DEFINED means. Typically this is by asserting an FIQ interrupt request input to the PE.

When an FIQ exception is taken, exception entry is precise to an instruction boundary.

As described in [Asynchronous exception masking controls](#), FIQ exceptions can be masked. When this happens, a generated FIQ exception is not taken until it is not masked.

By default, an FIQ exception is taken to FIQ mode, but an FIQ exception can be taken to EL2, that is Hyp mode.

For more information, see [The PE mode to which the physical FIQ exception is taken](#).

The preferred return address for an FIQ exception is the address of the instruction following the instruction boundary at which the exception was taken. For an exception taken to AArch32 state this return is performed as follows:

- If returning from a mode other than Hyp mode, using the [SPSR](#) and LR values generated by the exception entry, using an exception return instruction with a subtraction of 4. This means using:
 - [SPSR_fiq](#) and [LR_fiq](#) if returning from FIQ mode.
- If returning from Hyp mode, using the [SPSR_hyp](#) and [ELR_hyp](#) values generated by the exception entry, using an `ERET` instruction.

For more information, see [Exception return to an Exception level using AArch32](#).

D1.15.10.1 The PE mode to which the physical FIQ exception is taken

[Figure D1-8](#) shows how the implementation, state, and configuration options determine the PE mode to which an FIQ exception is taken when the exception is taken to an Exception level that is using AArch32.

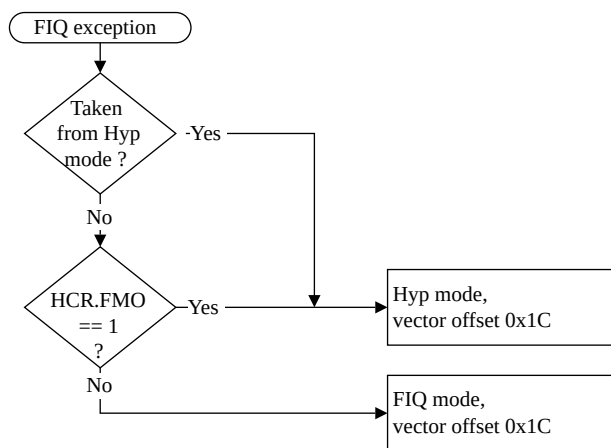


Figure D1-8 The PE mode the FIQ exception is taken to in AArch32 state

D1.15.10.2 Pseudocode description of taking the FIQ exception

The [AArch32_TakePhysicalFIQException\(\)](#) pseudocode procedure describes how the PE takes the exception.

D1.15.11 Virtual FIQ exception

The Virtual FIQ exception is implemented only as part of EL2, if EL2 is enabled in the current Security state.

A Virtual FIQ exception is generated in AArch32 state if all of the following apply:

- The PE is in a mode other than Hyp mode.
- The value of [PSTATE.F](#) is 0.
- The value of [HCR](#).{TGE, FMO} is {0, 1}.

- Either:
 - The value of [HCR.VF](#) is 1.
 - A Virtual FIQ exception is generated by an IMPLEMENTATION DEFINED mechanism.

The preferred return address for a Virtual FIQ exception is the address of the instruction immediately after the instruction boundary where the exception was taken. For an exception taken to AArch32 state this return is performed using the [SPSR](#) and [LR_irq](#) values generated by the exception entry, using an exception return instruction with a subtraction of 4.

D1.15.11.1 The PE mode to which the Virtual FIQ exception is taken

The Virtual FIQ exception is taken using a vector offset of $0 \times 1C$.

The conditions for generating a Virtual FIQ exception in AArch32 state mean the exception is:

- Taken from EL1 or EL0.
- Taken to FIQ mode if EL1 is using AArch32.

For more information, see [Virtual exceptions when an implementation includes EL2](#).

D1.15.11.2 Pseudocode description of taking the Virtual FIQ exception

The [AArch32_TakeVirtualFIQException\(\)](#) pseudocode procedure describes how the PE takes the exception.

D1.15.12 Additional pseudocode functions for exception handling

The [AArch32_EnterHypMode\(\)](#) pseudocode function changes the PE mode to Hyp mode, with the required state changes.

The [AArch32_EnterMode\(\)](#) pseudocode function changes the PE mode to a PL1 mode, with the required state changes. It is used for all exceptions that are not routed to Hyp mode or Monitor mode.

The [AArch32_EnterHypMode\(\)](#), and [AArch32_EnterMode\(\)](#) functions are described in [R-profile Architecture Pseudocode](#).

D1.16 Reset into AArch32 state

Resets and power domains describes the Armv8 reset model, including the defined levels of reset.

When reset is deasserted, the PE starts executing instructions at EL2 in Hyp mode with interrupts disabled.

Reset returns some PE state to architecturally-defined or IMPLEMENTATION DEFINED values, and makes other state UNKNOWN, as described in [PE state on reset into AArch32 state](#).

For more information about behavior when resetting into an Exception level using AArch32, see:

- [Behavior of caches at reset](#).
- [Reset and debug](#).

When reset is deasserted execution starts from an IMPLEMENTATION DEFINED address.

Software might be able to identify the reset address by reading [RVBAR](#).

The Arm architecture does not define any way of returning to a previous Execution state from a reset.

D1.16.1 PE state on reset into AArch32 state

Immediately after a reset, much of the PE state is architecturally UNKNOWN. However, some of the PE state is defined, see the individual register descriptions for more information. The state that is reset to known values is sufficient to permit predictable initial execution at the highest Exception level, such that this execution is then capable of initializing the remaining state of the system where necessary before use.

If the PE resets using either a Cold or a Warm reset, the PE state that is defined is as follows:

- The global exclusive monitor and local exclusive monitor for the PE are UNKNOWN.
- The PE resets into Hyp mode and PSTATE.M resets to 0b1010.
- [PSTATE.IL](#) resets to 0.

D1.16.2 Pseudocode descriptions of reset

The [AArch32_TakeReset](#) () pseudocode procedure describes how the PE behaves when reset is deasserted.

The [AArch32_ResetGeneralRegisters](#) () pseudocode function resets the general-purpose registers.

The [AArch32_ResetSIMDFPRegisters](#) () pseudocode function resets the SIMD and floating-point registers.

The [AArch32_ResetSpecialRegisters](#) () pseudocode function resets the Special-purpose registers, and the debug System registers [DLR](#) and [DSPSR](#), which are used for handling Debug exceptions.

The [AArch32_ResetSystemRegisters](#) () pseudocode function resets all System registers in the (coproc == 0b111x) encoding space to their reset state as defined in the register descriptions in [AArch32 System Register Descriptions](#).

Note

The [AArch32_ResetSystemRegisters](#) () function only resets the System registers. It has no effect on memory-mapped registers.

The [ResetExternalDebugRegisters](#) () pseudocode function resets all external debug registers to their reset state as defined in the register descriptions in [External Debug Register Descriptions](#).

D1.17 Mechanisms for entering a low-power state

The following sections describe the architectural mechanisms that a PE can use to request entry to a low-power state:

- [Wait For Event and Send Event](#).
- [Wait For Interrupt](#).

D1.17.1 Wait For Event and Send Event

The Wait For Event (WFE) mechanism permits a PE to request entry to a low-power state, and, if the request succeeds, to remain in that state until an event is generated by a Send Event operation, or another WFE wakeup event occurs.

[Example D1-2](#) describes how a spinlock implementation might use this mechanism to save energy.

Example D1-2 Spinlock as an example of using Wait For Event and Send Event

A multiprocessor operating system requires locking mechanisms to protect data structures from being accessed simultaneously by multiple PEs. These mechanisms prevent the data structures becoming inconsistent or corrupted if different PEs try to make conflicting changes. If a lock is busy, because a data structure is being used by one PE, it might not be practical for another PE to do anything except wait for the lock to be released. For example, if a PE is handling an interrupt from a device it might need to add data received from the device to a queue. If another PE is removing data from the queue, it will have locked the memory area that holds the queue. The first PE cannot add the new data until the queue is in a consistent state and the lock has been released. It cannot return from the interrupt handler until the data has been added to the queue, so it must wait.

Typically, a spin-lock mechanism is used in these circumstances:

- A PE requiring access to the protected data attempts to obtain the lock using single-copy atomic synchronization primitives such as the Load-Exclusive and Store-Exclusive operations described in [Synchronization and semaphores](#).
- If the PE obtains the lock, it performs its memory operation and releases the lock.
- If the PE cannot obtain the lock, it reads the lock value repeatedly in a tight loop until the lock becomes available. At this point, it again attempts to obtain the lock.

A spin-lock mechanism is not ideal for all situations:

- In a low-power system, the tight read loop is undesirable because it uses energy to no effect.
- In a multithreaded implementation, the execution of spin-locks by waiting threads can significantly degrade overall performance.

Using the Wait For Event and Send Event mechanism can improve the energy efficiency of a spinlock. In this situation, a PE that fails to obtain a lock can execute a Wait For Event instruction, `WFE`, to request entry to a low-power state. When a PE releases a lock, it must execute a Send Event instruction, `SEV`, causing any waiting PEs to wake up. Then, these PEs can attempt to gain the lock again.

The execution of a `WFE` instruction can cause suspension of execution only if all of the following are true:

- The instruction does not cause any other exception.
- When the instruction is executed:
 - The Event Register is not set.
 - There is not a pending WFE wakeup event.

For more information about the trap to EL2, see `HCR.{WFI, WFE}`.

The architecture does not define the exact nature of the low power state entered as a result of executing a `WFE` instruction, but the execution of a `WFE` instruction must not cause a loss of memory coherency or architectural state.

Note

Although a complex operating system can contain thousands of distinct locks, the event sent by this mechanism does not indicate which lock has been released. If the event relates to a different lock, or if another PE acquires the lock more quickly, the PE fails to acquire the lock and can reenter the low-power state waiting for the next event.

The Wait For Event system relies on hardware and software working together to achieve energy saving:

- The hardware provides the mechanism to enter the Wait For Event low-power state.
- The operating system software is responsible for issuing:
 - A Wait For Event instruction, to request entry to the low-power state, used in the example when waiting for a spin-lock.
 - A Send Event instruction, required in the example when releasing a spin-lock.

The mechanism depends on the interaction of:

- WFE wakeup events, see [WFE wakeup events](#).
- The Event Register, see [The Event Register](#).
- The Send Event instructions, see [The Send Event instructions](#).
- The Wait For Event instruction, see [The Wait For Event instruction](#).

D1.17.1.1 The Event Register

The Event Register is a single bit register for each PE. When set, an event register indicates that an event has occurred, since the register was last cleared, that might require some action by the PE. Therefore, the PE must not suspend operation on issuing a WFE instruction.

The reset value of the Event Register is UNKNOWN.

The Event Register for a PE is set by:

- The execution of an SEV instruction on any PE in the multiprocessor system.
- The execution of an SEVL instruction by the PE.
- An exception return.
- An event from a Generic Timer event stream, see [Event streams](#).
- An event sent by some IMPLEMENTATION DEFINED mechanism.

As shown in this list, the Event Register might be set by IMPLEMENTATION DEFINED mechanisms.

The Event Register is cleared only by a Wait For Event instruction.

Software cannot read or write the value of the Event Register directly.

D1.17.1.2 The Wait For Event instruction

The action of the Wait For Event instruction depends on the state of the Event Register:

- If the Event Register is 1, the instruction clears the register and completes immediately. Normally, if this happens the software makes another attempt to claim the lock.
- If the Event Register is 0, the PE can suspend execution, and hardware might enter a low-power state. The PE can remain suspended until a WFE wakeup event or a reset occurs. When a WFE wakeup event occurs, or earlier if the implementation chooses, the WFE instruction completes.

The execution of a WFE instruction that would otherwise cause suspension of execution might be trapped, see:

- [SCTLR](#).{nTWE, nTWI}.
- [HCR](#).{TWE, TWI}.

The Wait For Event instruction, WFE, is available at all privilege levels, see WFE.

Software using the Wait For Event mechanism must tolerate spurious wakeup events, including multiple wake ups.

D1.17.1.3 WFE wakeup events

The following events are *WFE wakeup events*:

- The execution of an `SEV` instruction on any PE in the system.
- The execution of an `SEVL` instruction on the PE.
- A physical IRQ interrupt, unless masked by the `PSTATE.I` bit.
- A physical FIQ interrupt, unless masked by the `PSTATE.F` bit.
- A physical SError exception, unless masked by the `PSTATE.A` bit.
- In any mode other than Hyp mode:
 - When `HCR.IMO` is set to 1, a virtual IRQ interrupt, unless masked by the `PSTATE.I` bit.
 - When `HCR.FMO` is set to 1, a virtual FIQ interrupt, unless masked by the `PSTATE.F` bit.
 - When `HCR.AMO` is set to 1, a virtual SError exception, unless masked by the `PSTATE.A` bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed*, see [Halting allowed and halting prohibited](#).
See also [External Debug Request debug event](#).
- An event sent by the timer event stream, see Event streams.
- An event sent by some IMPLEMENTATION DEFINED mechanism.
- An event caused by the clearing of the global monitor associated with the PE.

In addition to the possible masking of WFE wakeup events shown in this list, when invasive debug is enabled and `EDSCR.HDE` is set to 1, `EDSCR.INTdis` can mask interrupts, including masking them acting as WFE wakeup events. See the register description for more information.

As shown in the list of wakeup events, an implementation can include IMPLEMENTATION DEFINED hardware mechanisms to generate wakeup events.

Note

For more information about `PSTATE` masking, see [Asynchronous exception masking controls](#). If the configuration of the masking controls provided by EL2 mean that a `PSTATE` mask bit cannot mask the corresponding exception, then the physical exception is a WFE wakeup event, regardless of the value of the `PSTATE` mask bit.

D1.17.1.4 The Send Event instructions

The Send Event instructions are:

SEV, Send Event This causes an event to be signaled to all PEs in the multiprocessor system.

SEVL, Send Event Local

This must set the local Event Register. It might signal an event to other PEs, but is not required to do so.

The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED. The PE is not required to guarantee the ordering of this event with respect to the completion of memory accesses by instructions before the `SEV` instruction. Therefore, Arm recommends that software includes a `DSB` instruction before any `SEV` instruction.

Note

A `DSB` instruction ensures that no instruction, including any `SEV` instruction, that appears in program order after the `DSB` instruction, can execute until the `DSB` instruction has completed. For more information, see [Data Synchronization Barrier](#).

The `SEVL` instruction appears to execute in program order relative to any subsequent `WFE` instruction executed on the same PE, without the need for any explicit insertion of barrier instructions.

Execution of the Send Event instruction sets the Event Register.

The Send Event instructions are available at all privilege levels.

D1.17.1.5 Pseudocode description of the Wait For Event mechanism

This section defines pseudocode functions that describe the operation of the Wait For Event mechanism.

The `ClearEventRegister()` pseudocode procedure clears the Event Register of the current PE.

The `IsEventRegisterSet()` pseudocode function returns TRUE if the Event Register of the current PE is 1 and FALSE if it is 0.

The `WaitForEvent()` pseudocode procedure optionally suspends execution until a WFE wakeup event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes a `ClearEventRegister()` to occur.

The `SendEvent()` pseudocode procedure sets the Event Register of every PE in the system.

D1.17.2 Wait For Interrupt

AArch32 state supports Wait For Interrupt through an instruction, `WFI`, which is provided in the T32 and A32 instruction sets. For more information, see `WFI`.

When a PE issues a `WFI` instruction, its execution can be suspended, and a low-power state can be entered.

The execution in AArch32 state of a `WFI` instruction that would otherwise cause suspension of execution might be trapped, see:

- `SCTLR.nTWI`.
- `HCR.WFI`.

The execution of a `WFI` instruction can cause suspension of execution only if both:

- The instruction does not cause any other exception.
- When the instruction is executed, there is not a pending WFI wakeup event.

D1.17.2.1 WFI wakeup events

The PE can remain suspended in its WFI state until it is reset, or one of the following *WFI wakeup events* occurs:

- A physical IRQ interrupt, regardless of the value of the `PSTATE.I` bit.
- A physical FIQ interrupt, regardless of the value of the `PSTATE.F` bit.
- A physical SError exception, regardless of the value of the `PSTATE.A` bit.
- In any mode other than Hyp mode:
 - When `HCR.IMO` is set to 1, a virtual IRQ interrupt, regardless of the value of the `PSTATE.I` bit.
 - When `HCR.FMO` is set to 1, a virtual FIQ interrupt, regardless of the value of the `PSTATE.F` bit.
 - When `HCR.AMO` is set to 1, a virtual SError exception, regardless of the value of the `PSTATE.A` bit.
- An asynchronous External Debug Request debug event, if halting is allowed. For the definition of *halting is allowed*, see [Halting allowed and halting prohibited](#).
See also [External Debug Request debug event](#).

An implementation can include other IMPLEMENTATION DEFINED hardware mechanisms to generate WFI wakeup events.

When a WFI wakeup event is detected, or earlier if the implementation chooses, the `WFI` instruction completes.

WFI wakeup events cannot be masked by the mask bits in the `PSTATE`.

The architecture does not define the exact nature of the low power state, except that:

- The execution of a `WFI` instruction must not cause a loss of memory coherency.
- If the system is configured such that the `WFI` instruction can be completed, then the `WFI` instruction must not cause a loss of architectural state.

Note

In some implementations, based on the configuration of system specific registers, WFI can be used as part of a powerdown sequence where no interrupts will cause WFI wakeup events, and restoration of power involves resetting of the PE. In those cases, the WFI is permitted to cause a loss of architectural state, as it is assumed that this state will have been saved by software as part of the powerdown sequence before the WFI.

Note

- Because debug events are WFI wakeup events, Arm strongly recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures the intervention of debug while waiting does not significantly change the function of the program being debugged.
 - In some previous implementations of Wait For Interrupt, the idle loop is followed by exit functions that must be executed before taking the interrupt. The operation of Wait For Interrupt remains consistent with this model, and therefore differs from the operation of Wait For Event.
 - Some implementations of Wait For Interrupt drain down any pending memory activity before suspending execution. The Arm architecture does not require this operation, and software must not rely on Wait For Interrupt operating in this way.
-

D1.17.2.2 Using WFI to indicate an idle state on bus interfaces

A common implementation practice is to complete any entry into powerdown routines with a `WFI` instruction. Typically, the `WFI` instruction:

1. Forces the completion of execution of any instructions that are in progress, and of all associated bus activity.
2. Suspends the execution of instructions by the PE.

The control logic required to do this tracks the activity of the bus interfaces used by the PE. This means it can signal to an external power controller when there is no ongoing bus activity.

However, memory-mapped and external debug interface accesses to debug registers must continue to be processed while the PE is in the WFI state. The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

If [FEAT_DoubleLock](#) is implemented and the value of `DBGOSDLR.DLK`, the OS Double Lock status bit, is set to 1, this idle state must not be signaled to the PE unless the system can guarantee, also, that the debug interface is idle.

Note

When separate Core and Debug power domains are implemented, the debug interface referred to in this section is the interface between the Core and Debug power domains, since the signal to the power controller indicates that the Core power domain is idle. For more information about the power domains, see [Power domains and debug](#).

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred powerdown entry mechanism.

D1.17.2.3 Pseudocode description of Wait For Interrupt

The `WaitForInterrupt()` pseudocode function optionally suspends execution until a WFI wakeup event or reset occurs, or until some earlier time if the implementation chooses.

D1.18 The AArch32 System register interface

Most System registers are accessed using the instructions described in [System register access instructions](#). The System register interface provides access to those instructions, and:

- These registers are encoded using the parameters {coproc, opc1, CRn, CRm, opc2}, with permitted coproc values of 0b1110 and 0b1111.
- Some of these encodings provide the AArch32 System instructions.
- To maintain compatibility with previous versions of the Arm architecture, the access controls for the AArch32 System registers include the access controls for AArch32 Advanced SIMD and floating-point functionality.

Note

See [Background to the System register interface](#) for more information.

The following sections give more information about the AArch32 System register interface:

- [System registers in the coproc == 0b111x encoding space](#).
- [Access to System registers](#).
- [Access controls for Advanced SIMD and floating-point functionality](#).
- [Background to the System register interface](#).

D1.18.1 System registers in the coproc == 0b111x encoding space

In AArch32 state:

- The coproc == 0b1110 encoding space is reserved for the configuration and control of:
 - Debug features, see [Debug System registers](#).
 - Trace features, see the *Embedded Trace Macrocell Architecture Specification*.
 - Identification registers for the Trivial Jazelle implementation, see [Trivial implementation of the Jazelle extension](#).
- The coproc == 0b1111 encoding space is reserved for the control and configuration of the PE, including architecture and feature identification. This means these encodings provide access to the System registers that control and return status information for PE operation.

For more information, see [AArch32 System Register Descriptions](#).

D1.18.2 Access to System registers

Most System registers are accessible only from EL1 or higher. For possible accesses from EL0 the register descriptions in [AArch32 System Register Descriptions](#) indicate whether a register is accessible from EL0.

D1.18.3 Access controls for Advanced SIMD and floating-point functionality

The CPACR controls access to Advanced SIMD and floating-point functionality from software executing at PL1 or EL0 in AArch32 state:

- The {cp10, cp11} fields control access to all Advanced SIMD and floating-point functionality, and can:
 - Disable EL0 and PL1 access to this functionality.
 - Enable access to this functionality at PL1 only.
 - Enable access to this functionality at EL0 and PL1.
- The ASEDIS field controls access to Advanced SIMD instructions that are not also floating-point instructions.

Initially on powerup or reset into AArch32 state, all access to all Advanced SIMD and floating-point functionality from PL1 and EL0 is disabled.

Note

The [CPACR](#) has no effect on accesses from Hyp mode.

[HCPTR](#) provides additional controls on accesses to Advanced SIMD and floating-point functionality. For accesses that are otherwise permitted by the [CPACR](#) and [NSACR](#) settings, setting [HCPTR](#) bits to 1:

- Traps otherwise-permitted accesses from EL1 or EL0 to EL2. When EL2 is using AArch32, these accesses are trapped to Hyp mode.
- Makes accesses from EL2 mode UNDEFINED. When EL2 is using AArch32, this makes accesses from Hyp mode UNDEFINED.

In the [HCPTR](#):

- The {TCP10, TCP11} fields control access to all Advanced SIMD and floating-point functionality.
- The TASE field controls access to Advanced SIMD instructions that are not also floating-point instructions.
- The TCPAC field traps EL1 accesses to the [CPACR](#) to Hyp mode.

Note

Whenever a pair of fields control the access to the Advanced SIMD and floating-point functionality, the values of each field of the pair must be identical. If these settings are not identical the behavior of the Advanced SIMD and floating-point functionality is CONstrained UNPREDICTABLE, see [Handling of System register control fields for Advanced SIMD and floating-point operation](#).

For more information about Advanced SIMD and floating-point support, see [Advanced SIMD and floating-point support](#).

D1.18.4 Background to the System register interface

Note

This section is not part of the Architecture specification. It is included only to present the rationale of some aspects of the System register interface.

The interface to the System registers was originally defined as part of a generic *coprocessor* interface that gave access to 15 coprocessors, CP0 - CP15. Of these, CP8 - CP15 were reserved for use by Arm, while CP0 - CP7 were available for IMPLEMENTATION DEFINED coprocessors.

The coprocessors were accessed using coprocessor instructions. These instructions remain part of the T32 and A32 instruction sets, see [System register access instructions](#).

In the Arm coprocessor model, a coprocessor included both:

- Primary and secondary coprocessor registers, which form part of the coprocessor interface.
- A number of internal registers.

When accessing a 32-bit internal coprocessor register, using an MCR or MRC instruction, the instruction specified:

- The target coprocessor, specified by the `coproc` parameter and taking a value between p0 for CP0 and p15 for CP15.
- The primary coprocessor register, specified by the `CRn` parameter and taking a value between c0 and c15.
- The secondary coprocessor register, specified by the `CRm` parameter and taking a value between c0 and c15.
- Up to two additional parameters, `opc1` and `opc2`, taking values between 0 and 7.

Other instructions in the group described in [System register access instructions](#) take a subset of these parameters:

- MCRR and MRRC instructions take parameters {`coproc`, `opc1`, `CRm`}, where `CRm` is the primary coprocessor register.

To maintain backward compatibility, the arguments to an MCR or MRC instruction remain {coproc, opc1, CRn, CRm, opc2}. However:

- The naming of these parameters no longer has any particular significance.
- While the coproc field is a 4-bit field, opc1 is a 2-bit field.

Of the coprocessors reserved for use by Arm, in earlier versions of the architecture:

- CP15 provided access to the System registers relating to non-debug operation, and was originally called the System control coprocessor. From the introduction of Armv8, these registers are described as being in the coproc == 0b1111 encoding space.
- CP14 provided access to additional System registers, including those relating to debug and trace. From the introduction of Armv8, these registers are described as being in the coproc == 0b1110 encoding space.
- CP10 and CP11 were used for Advanced SIMD and floating-point control, and many coprocessor instruction encodings targeting CP10 and CP11 were used as floating-point instruction encodings:
 - Generally from the introduction of Armv8 the architecture does not relate these instructions to the coprocessor encoding space, but the naming of registers and register fields for Advanced SIMD and floating-point control reflects the historic coprocessor model.
 - Because the Advanced SIMD and floating-point functionality used both CP10 and CP11, some System register controls of this functionality have a pair of fields, for example NSACR.{cp10, cp11}. In these cases, both fields must be set to the same value. For more information, see [Access controls for Advanced SIMD and floating-point functionality](#).

From the introduction of Armv8:

- The AArch32 System registers include registers that were described as Special registers in Armv7 and earlier versions of the architecture. This means that the System registers include registers that are outside the earlier coprocessor model.
- The Armv7 AArch32 instruction encodings for LDC, STC, MCR, MRC, MCRR, and MRRC instructions with coproc field values other than {1010, 1011, 1110, 1111} are available for reuse. Armv8.2 re-uses some encodings in this way.

D1.19 Advanced SIMD and floating-point support

[Advanced SIMD and floating-point instructions](#) introduces:

- The scalar floating-point instructions in the T32 and A32 instruction sets.
- The Advanced SIMD integer and floating-point vector instructions in the T32 and A32 instruction sets.
- The SIMD and floating-point register file, which can be viewed as:
 - Singleword registers S0 - S31.
 - Doubleword registers D0 - D31.
 - Quadword registers Q0 - Q15.
- The *Floating-Point Status and Control Register (FPSCR)*.
- [Floating-point exceptions and exception traps](#).

For more information about the System registers for the Advanced SIMD and floating-point operation, see [Advanced SIMD and floating-point System registers](#). Software can interrogate these registers to discover the implemented Advanced SIMD and floating-point support.

[AArch32 implications of not including support for Advanced SIMD and floating-point](#) summarizes the effects of not supporting these instructions, and the following subsections give more information about the Advanced SIMD and Floating-point support:

- [Enabling Advanced SIMD and floating-point support](#).
- [Advanced SIMD and floating-point System registers](#).
- [Context switching when using Advanced SIMD and floating-point functionality](#).

D1.19.1 AArch32 implications of not including support for Advanced SIMD and floating-point

The architecture generally requires the inclusion of the Advanced SIMD and floating-point instructions in all instruction sets. However, for implementations targeting specialized markets, Arm might produce or license implementations that do not provide any support for Advanced SIMD and floating-point instructions. In such an implementation, in AArch32 state:

- The [CPACR](#).{ASEDIS, cp11, cp10} fields are RES0.
- The [NSACR](#).{NSASEDIS, cp11, cp10} fields are RES0.
- The [HCPTR](#).{TASE, TCP11, TCP10} fields are RES1.
- The [FPExc](#), [FPSCR](#), [FPSID](#), [MVFR0](#), [MVFR1](#), and [MVFR2](#) registers are not implemented and their encodings are UNDEFINED.
- Attempted accesses to Advanced SIMD and floating-point functionality are UNDEFINED. This means:
 - All Advanced SIMD and floating-point instructions are UNDEFINED.
 - Attempts to access the Advanced SIMD and floating-point System registers are UNDEFINED.

D1.19.2 Enabling Advanced SIMD and floating-point support

Software must ensure that the required access to the Advanced SIMD and floating-point features is enabled. Most of those controls are described in [Configurable instruction controls](#), and this section:

- Summarizes those controls.
- Provides additional information in the following subsections:
 - [FPExc control of access to Advanced SIMD and floating-point functionality](#).
 - [EL0 access to Advanced SIMD and floating-point functionality](#).

The controls of access to Advanced SIMD and floating-point functionality are:

General {cp10, cp11} or {TCP10, TCP11} controls

This relates to the [CPACR](#).{cp10, cp11}, [NSACR](#).{cp10, cp11}, and [HCPTR](#).{TCP10, TCP11} controls.

Note

[Background to the System register interface](#) explains the naming of these controls.

The {cp10, cp11} controls provide general control of the use of Advanced SIMD and floating-point functionality, as follows:

- [CPACR](#).{cp10, cp11} control access from PE modes other than Hyp mode. These fields have no effect on accesses to Advanced SIMD and floating-point functionality from Hyp mode.
- [HCPTR](#).{TCP10, TCP11} provides controls on accesses.

In each case, the {cp10, cp11} controls must be programmed to the same value, otherwise operation is CONSTRAINED UNPREDICTABLE. The CONSTRAINED UNPREDICTABLE behavior is that, for all purposes other than reading the value of the register field, behavior is as if the cp11 field has the same value as the cp10 field. For more information, see [Handling of System register control fields for Advanced SIMD and floating-point operation](#).

For more information about these controls, see:

- [FPEXC](#).
- [CPACR](#).
- [HCPTR](#).
- [NSACR](#).

Control of accesses to the CPACR from PL1 modes

As stated in [General {cp10, cp11} or {TCP10, TCP11} controls](#), the [CPACR](#) controls access to Advanced SIMD and floating-point functionality from PE modes other than Hyp mode. Accesses to the [CPACR](#) from PL1 modes can be trapped to EL2.

Additional controls of Advanced SIMD functionality

- If implemented as an RW field, [CPACR](#).ASEDIS can make all Advanced SIMD instructions UNDEFINED in all modes other than Hyp mode.
- In an implementation that includes EL2, when the [CPACR](#) and [NSACR](#) settings permit use of the Advanced SIMD instructions, if [HCPTR](#).TASE is implemented as an RW field it can make these instructions UNDEFINED in Hyp mode, and trap to Hyp mode any use of these instructions in a PL0 or PL1 mode.

[Pseudocode description of enabling SIMD and floating-point functionality](#) provides links to the pseudocode descriptions of all of these controls.

D1.19.2.1 FPEXC control of access to Advanced SIMD and floating-point functionality

In addition, [FPEXC](#).EN is an enable bit for most Advanced SIMD and floating-point operations. When [FPEXC](#).EN is 0, all Advanced SIMD and floating-point instructions are treated as UNDEFINED except for:

- A VMSR to the [FPEXC](#) or [FPSID](#) register.
- A VMRS from the [FPEXC](#), [FPSID](#), [MVFR0](#), [MVFR1](#), or [MVFR2](#) register.

These instructions can be executed only at EL1 or higher.

Note

- When the [FPSID](#) is accessible, any write access to the [FPSID](#) is ignored.
- When [FPEXC](#).EN is 0, these operations are treated as UNDEFINED :

- A VMSR to the [FPSCR](#).
 - A VMRS from the [FPSCR](#).
-

D1.19.2.2 EL0 access to Advanced SIMD and floating-point functionality

When the access controls summarized in this section permit EL0 access to the Advanced SIMD and floating-point functionality, this applies only to the subset of functionality that is available at EL0. In particular:

- Only Advanced SIMD and Floating-point System register that is accessible is the [FPSCR](#).
- The Advanced SIMD and floating-point instructions are available.

Execution at EL0 corresponds to the application level view of the Advanced SIMD and floating-point functionality, as described in [Advanced SIMD and floating-point System registers](#).

D1.19.3 Advanced SIMD and floating-point System registers

AArch32 state provides a common set of System registers for the Advanced SIMD and floating-point functionality. This section gives general information about this set of registers, and indicates where each register is described in detail. It contains the following subsections:

- [Register map of the Advanced SIMD and floating-point System registers](#).
- [Accessing the Advanced SIMD and floating-point System registers](#).

D1.19.3.1 Register map of the Advanced SIMD and floating-point System registers

[Table D1-14](#) shows the register map of the Advanced SIMD and Floating-point registers. Each register is 32 bits wide.

Table D1-14 Floating-point registers

Name	Permitted access
FPEXC	RW
FPSCR	RW
FPSID	RW ^a
MVFR0	RO
MVFR1	RO
MVFR2	RO

a. When [FPSID](#) is accessible, [VMSR](#) accesses to [FPSID](#) are ignored.

D1.19.3.2 Accessing the Advanced SIMD and floating-point System registers

Software accesses the Advanced SIMD and floating-point System registers using the [VMRS](#) and [VMSR](#) instructions, see:

- [VMRS](#).
- [VMSR](#).

For example:

```

VMRS <Rt>, FPSID    ; Read Floating-Point System ID Register
VMRS <Rt>, MVFR1     ; Read Media and VFP Feature Register 1
VMSR FPSCR, <Rt>    ; Write Floating-Point System Control Register

```

Software can access the Advanced SIMD and floating-point System registers only if the access controls permit the access, see [Enabling Advanced SIMD and floating-point support](#).

Note

All hardware ID information can be accessed only from EL1 or higher. This means:

The FPSID is accessible only from EL1 or higher.

This is a change introduced from VFPv3. Previously, the [FPSID](#) register can be accessed in all modes.

The MVFR registers are accessible only from EL1 or higher.

Unprivileged software must issue a system call to determine what features are supported.

D1.19.4 Context switching when using Advanced SIMD and floating-point functionality

When the Advanced SIMD and floating-point functionality is used by only a subset of processes, the operating system might implement lazy context switching of the Advanced SIMD and floating-point register file and System registers.

In the simplest lazy context switch implementation, the primary context switch software uses the [CPACR](#).{cp10, cp11} controls to disable access to the Advanced SIMD and floating-point functionality, see [Enabling Advanced SIMD and floating-point support](#). Subsequently, when a process or thread attempts to use an Advanced SIMD or floating-point instruction, it triggers an Undefined Instruction exception. The operating system responds by saving and restoring the Advanced SIMD and floating-point register file and System registers. Typically, it then re-executes the Advanced SIMD or floating-point instruction that generated the Undefined Instruction exception.

D1.20 Configurable instruction controls

This section describes the *Configurable instruction controls* provided by the AArch32 state.

Each control is categorized as an *instruction enable*, an *instruction disable*, or a *trap control*. The definitions of each type overlap, and in some cases are historical. Each configurable instruction control defines how an exception that is generated as a consequence of the configurable instruction control is handled or reported.

Instruction enables and instruction disables

Enable or disable the use of one or more particular instructions at a particular Privilege level and Security state.

When an instruction is disabled as a result of an instruction enable or disable, it is UNDEFINED.

The exception generated by attempting to execute an UNDEFINED instruction is:

- Taken to EL1 if the UNDEFINED instruction was executed at EL0, unless the instruction was executed at EL0 and is routed to EL2 by the control described in [Routing exceptions from EL0 to EL2](#).
When the exception is taken to EL1, it is taken to Undefined mode.
- Otherwise, taken to the Exception level at which the UNDEFINED instruction was executed:
 - If the instruction was executed in Hyp mode the exception is taken to Hyp mode.
 - Otherwise, the exception is taken to Undefined mode.

Trap controls

Control whether one or more instructions, when executed at a particular Privilege level, are *trapped*.

Trap controls are grouped as:

PL1

Trapped instructions generate Undefined Instruction exceptions that are taken to Undefined mode, unless the instruction was executed at EL0 and is routed to EL2 by the control described in [Routing exceptions from EL0 to EL2](#).

For more information about these traps, see [PL1 configurable controls](#).

Hyp mode (PL2)

Trapped instructions generate:

- Hyp Trap exceptions, taken to Hyp mode, if trapped from a mode other than Hyp mode.
- Undefined Instruction exceptions taken to Hyp mode, if trapped from Hyp mode.

For more information about these traps, see [EL2 configurable controls](#).

See also [Routing exceptions from EL0 to EL2](#).

An exception generated as a result of an instruction enable or disable, or a trap control, is only taken if the instruction does not also generate a higher priority exception. [Exception prioritization for exceptions taken to AArch32 state](#) defines the prioritization of different exceptions on the same instruction.

Exceptions generated as a result of these controls are synchronous exceptions.

For exceptions taken to an Exception level that is using AArch32, only exceptions that are taken to Hyp mode are reported in a syndrome register, the [HSR](#).

An implementation might provide additional controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.

When a configurable instruction control causes an exception, the exception is taken and the instruction is not executed, and therefore all the following are true:

- The preferred exception return address of the exception is the instruction that generates the exception.

- There are no changes to the registers accessed by the instruction, including as a result of side-effects of a register access.

This section is organized as follows:

- [Instructions that fail their Condition code check.](#)
- [Instructions that are UNPREDICTABLE.](#)
- [Register access instructions.](#)
- [PL1 configurable controls.](#)
- [EL2 configurable controls.](#)
- [Pseudocode description of configurable instruction enables, disables, and traps.](#)

D1.20.1 Instructions that fail their Condition code check

When a configurable instruction control causes a conditional instruction to generate an exception in AArch32 state, it is IMPLEMENTATION DEFINED whether the exception applies to conditional AArch32 instructions that fail their condition code check.

For instructions that fail their Condition code check, see [Conditional execution of undefined instructions](#).

For an instruction that has a Hyp trap set that fails its Condition code check:

- Unless the trap description states otherwise, it is IMPLEMENTATION DEFINED whether the instruction:
 - Generates a Hyp Trap exception.
 - Executes as a NOP.

Any implementation must be consistent in its handling of instructions that fail their Condition code check. This means that:

- Whenever a Hyp trap is set on an instruction it must either:
 - Always generate a Hyp Trap exception.
 - Always treat the instruction as a NOP.
- The IMPLEMENTATION DEFINED part of the requirements of [Conditional execution of undefined instructions](#) must be consistent with the handling of Hyp traps on instructions that fail their Condition code check. [Table D1-15](#) shows this:

Table D1-15 Consistent handling of instructions that fail their Condition code check

Behavior of conditional undefined instruction ^a	Hyp trap on instruction that fails its Condition code check ^b
Executes as a NOP	Executes as a NOP
Generates an Undefined Instruction exception	Generates a Hyp Trap exception

a. As defined in [Conditional execution of undefined instructions](#). In EL0 and EL1 modes, this applies only if no Hyp trap is set for the instruction, otherwise see the behavior in the other column of the table.

b. For a trapped instruction executed in an EL1 or EL0 mode.

D1.20.2 Instructions that are UNPREDICTABLE

It is UNPREDICTABLE / CONSTRAINED UNPREDICTABLE whether configurable instruction controls generate an exception when the instruction is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed, except in the following cases:

- If the instruction description explicitly states that the configurable instruction control is applied with higher priority than the CONSTRAINED UNPREDICTABLE behavior, then the configurable instruction control generates an exception.

- For an instruction where one of the CONSTRAINED UNPREDICTABLE behaviors is that a register value becomes UNKNOWN, then either:
 - The configurable instruction controls generate an exception.
 - The instruction executes with one of the CONSTRAINED UNPREDICTABLE behaviors that does not cause a register value to become UNKNOWN.

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions can generate exceptions as a result of configurable instruction controls, but the architecture does not require them to do so.

Note

UNPREDICTABLE and CONSTRAINED UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower Exception level using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. This means that disabling or trapping an instruction changes the set of instructions that might be executed at EL1 or EL0, or in Hyp mode. This indirectly affects the permitted behavior of UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions. Many CONSTRAINED UNPREDICTABLE behaviors for instructions include an allowance that the CONSTRAINED UNPREDICTABLE instruction behaves the same way as a closely related instruction that is not CONSTRAINED UNPREDICTABLE. In those cases, the instruction enable, disable, or trap control that causes an exception on the closely related instruction will cause the same exception on the CONSTRAINED UNPREDICTABLE instruction.

If no instructions are trapped, the attempted execution of an UNPREDICTABLE instruction in an EL1 or EL0 mode must not generate a Hyp Trap exception.

D1.20.3 Register access instructions

When an instruction is disabled or trapped, the exception is taken before execution of the instruction. This means that if the instruction is a register access instruction:

- No access is made before the exception is taken.
- Side-effects that are normally associated with the access do not occur before the exception is taken.

D1.20.4 PL1 configurable controls

Any exception that is generated because of an AArch32 PL1 control is taken to a PL1 mode.

Note

Any exception generated because of an AArch32 PL1 control is taken to AArch32 state.

[Table D1-16](#) shows the AArch32 System registers that contain these controls.

Table D1-16 PL1 System registers that contain configurable instruction controls

Register name	Register description
SCTLR	System Control Register
FPEXC	Floating-point Exception Control Register
CPACR	Architectural Feature Access Control Register
DBGDSCRExt	Monitor System Debug Control Register
PMUSERENR	Performance Monitors User Enable Register
CNTKCTL	Counter-timer Kernel Control Register

When generated in User mode, exceptions generated by these controls can be routed to EL2, as described in [Routing exceptions from EL0 to EL2](#).

D1.20.5 EL2 configurable controls

Table D1-17 shows the System registers that contain these controls.

Table D1-17 EL2 System registers that contain configurable instruction controls

Register name	Register description
FPEXC	Floating-point Exception Control Register
HCR	Hypervisor Configuration Register
HSTR	Hypervisor System Trap Register
HCPTR	Hyp Architectural Feature Trap Register
HDCR	Hyp Debug Control Register
HSCTLR	Hyp System Control Register
HCR2	Hypervisor Configuration Register 2
CNTHCTL	Counter-timer Hyp Control Register

D1.20.6 Pseudocode description of configurable instruction enables, disables, and traps

The pseudocode function [AArch32_CheckITEnabled\(\)](#) checks whether the T32 IT instruction is enabled.

The pseudocode function [AArch32_CheckSETENDEnabled\(\)](#) checks whether the SETEND instruction is disabled.

The pseudocode function for [AArch32_CheckForSMCUndefOrTrap\(\)](#) checks for traps on an SMC instruction.

The [AArch32_CheckForWFXTrap\(\)](#) pseudocode function checks for traps on WFE and WFI instructions:

D1.20.6.1 Pseudocode description of enabling SIMD and floating-point functionality

The [AArch32_CheckAdvSIMDOrFPEEnabled\(\)](#) and [AArch32_CheckFPAdvSIMDTrap\(\)](#) pseudocode functions take appropriate action if an SIMD or floating-point instruction is used when the SIMD and floating-point functionality is not enabled or is trapped.

The [CheckAdvSIMDOrVFPEEnabled\(\)](#), [CheckAdvSIMDEnabled\(\)](#), and [CheckVFPEEnabled\(\)](#) wrapper functions support the [AArch32_CheckAdvSIMDOrFPEEnabled\(\)](#) and [AArch32_CheckFPAdvSIMDTrap\(\)](#) functions.

The [AArch32_CheckAdvSIMDOrFPEEnabled\(\)](#), [AArch32_CheckFPAdvSIMDTrap\(\)](#), [CheckAdvSIMDOrVFPEEnabled\(\)](#), [CheckAdvSIMDEnabled\(\)](#), and [CheckVFPEEnabled\(\)](#) functions are described in [A-profile Architecture Pseudocode](#).

Chapter D2

AArch32 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch32 self-hosted debug Exception model. It is organized as follows:

- [About self-hosted debug](#)
- [Routing debug exceptions](#)
- [The debug exception enable controls](#)
- [The effect of powerdown on debug exceptions](#)
- [Summary of permitted routing and enabling of debug exceptions](#)
- [Pseudocode description of debug exceptions](#)
- [Breakpoint Instruction exceptions](#)
- [Breakpoint exceptions](#)
- [Watchpoint exceptions](#)
- [Vector Catch exceptions](#)
- [Synchronization and debug exceptions](#)

D2.1 About self-hosted debug

Self-hosted debug supports debugging through the generation and handling of *debug exceptions*, that are taken using the Exception model described in [The AArch32 System Level Programmers' Model](#).

This section introduces some terms used in describing self-hosted debug, and then introduces the debug exceptions. See:

- [Definition of a debugger in the context of self-hosted debug](#).
- [Context ID and Process ID](#).

D2.1.1 Definition of a debugger in the context of self-hosted debug

Within this chapter, *debugger* means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the debug System registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective, the debug services are the debugger.

D2.1.2 Context ID and Process ID

The [CONTEXTIDR](#) identifies the current *Context ID*, that is used by:

- The debug logic, for breakpoint and watchpoint matching.
- Implemented trace logic, to identify the current process.

The [CONTEXTIDR](#) has a single field, PROCID, that is defined as the *Process Identifier* (Process ID).

D2.1.3 About debug exceptions

Debug exceptions occur during normal program flow if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions. The debug exceptions that can be generated in a stage 1 translation regime are:

- [Breakpoint Instruction exceptions](#).
- [Breakpoint exceptions](#), generated by hardware breakpoints.
- [Watchpoint exceptions](#), generated by hardware watchpoints.
- [Vector Catch exceptions](#).

Other than Breakpoint Instruction exceptions, the PE can generate a particular debug exception only if all of the following are true:

- The OS Lock is unlocked.
- `DoubleLockStatus()` == FALSE.
- Debug exceptions are not disabled from the current Exception level.
See [Enabling debug exceptions](#). Breakpoint Instruction exceptions are always enabled from the current Exception level.
- A debugger has not disabled that particular debug exception.
All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the [DBGDSCRExt](#). See [The debug exception enable controls](#).

Breakpoint Instruction exceptions are always enabled.

Note

If *halting is allowed* and [EDSCR.HDE](#) is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.
For the definition of halting is allowed, see [Halting allowed and halting prohibited](#).

When a debug exception is taken:

- If the debug exception is a Watchpoint exception, it is taken as a Data Abort exception.
- Otherwise, it is taken as a Prefetch Abort exception.

The following list summarizes each of the debug exceptions:

Breakpoint Instruction exceptions

Breakpoint instructions generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction in the T32 and A32 instruction sets is `BKPT #<immediate>`. Whenever one of these is committed for execution, the PE takes a Breakpoint Instruction exception.

PE behavior

Breakpoint Instruction exceptions cannot be masked. The PE takes Breakpoint Instruction exceptions regardless of both of the following:

- The current Privilege level and AArch32 mode.
- The current Security state.

For more information, see [Breakpoint Instruction exceptions](#).

Breakpoint exceptions

The architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on one or more of instruction addresses, accesses to any address in an instruction address range, or particular PE contexts.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address `0x1000` is committed for execution.

The architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch32 translation regime:

- Address:
 - Address Match.
 - Address Mismatch.Comparisons are made with the virtual address of each instruction in the program flow.
- Context:
 - Context ID Match. Matches with the Context ID value held in the [CONTEXTIDR](#).
 - VMID Match. Matches with the VMID value held in the [VSCTLR](#).
 - Context ID and VMID Match. Matches with both the Context ID and the VMID value.

An Address breakpoint can link to a Context breakpoint, so that the Address breakpoint only generates a Breakpoint exception if the PE is in a particular context when the address match or mismatch occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

PE behavior

If halting is allowed and [EDSCR.HDE](#) is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See [Debug State](#).

Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see [Breakpoint exceptions](#).

Watchpoint exceptions

The architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range $0 \times 1000 - 0 \times 101F$.

A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is a Context matching breakpoint with linking enabled, or it can link to an address breakpoint. In this case, the watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

PE behavior

If halting is allowed and `EDSCR.HDE` is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See [Debug State](#).

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see [Watchpoint exceptions](#).

Vector Catch exceptions

These are used to trap exceptions. The architecture provides two forms of vector catch, *address-matching* and *exception-trapping*. Only one form can be implemented.

Whichever form is implemented, a debugger must enable Vector Catch exceptions for one or more exception vectors by programming the `DBGVCR`. Generation of Vector Catch exceptions is then as follows:

- For the address-matching form, a Vector Catch exception is generated whenever the virtual address of an instruction matches a vector that Vector Catch exceptions are enabled for.
- For the Exception-trapping form, a Vector Catch exception is generated as part of exception entry for exception types that correspond to vectors that Vector Catch exceptions are enabled for.

PE behavior

If debug exceptions are:

- Enabled, Vector Catch exceptions can be generated.
- Disabled, vector catch is ignored.

For more information, see [Vector Catch exceptions](#).

[Table D2-1](#) summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

Table D2-1 PE behavior and pseudocode for each of the debug exceptions

Debug exception	PE behavior if debug exceptions are:		Pseudocode, see
	Enabled	Disabled	
Breakpoint Instruction exception	Takes Prefetch Abort exception	Takes Prefetch Abort exception	AArch32_SoftwareBreakpoint()
Breakpoint exception	Takes Prefetch Abort exception ^a	Ignored	Pseudocode description of Breakpoint exceptions taken from AArch32 state
Watchpoint exception	Takes Data Abort exception ^a	Ignored	Pseudocode description of Watchpoint exceptions taken from AArch32 state
Vector Catch exception	Takes Prefetch Abort exception	Ignored	Pseudocode description of Vector Catch exceptions

a. If halting is allowed and [EDSCR.HDE](#) is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See [Debug State](#).

D2.2 Routing debug exceptions

Debug exceptions are usually routed to Abort mode. However, the routing of debug exceptions depends on the [Effective values](#) of [HDCR.TDE](#) and [HCR.TGE](#):

If the [Effective value](#) of {[HDCR.TDE](#), [HCR.TGE](#)} is not {0, 0}

Debug exceptions are routed to Hyp mode.

Otherwise

Debug exceptions behave as follows:

- Debug exceptions taken from EL1 and EL0 are routed to Abort mode.
- Breakpoint Instruction exceptions taken from Hyp mode are routed to Hyp mode.
- All other debug exceptions are disabled from Hyp mode.

[Table D2-2](#) shows the routing of debug exceptions taken from an Exception level that is using AArch32 to an Exception level that is using AArch32. In these tables:

TDE Means the logical OR of [HDCR.TDE](#) and [HCR.TGE](#).

(Hyp mode) Means:

- All debug exceptions other than Breakpoint Instruction exceptions are disabled from this Privilege level.
- Breakpoint Instruction exceptions taken from this Privilege level are taken to Hyp mode.

Table D2-2 Routing of debug exceptions

TDE	Target AArch32 mode:		
	PL0	PL1	PL2
0	Abort mode	Abort mode	(Hyp mode)
1	Hyp mode	Hyp mode	(Hyp mode)

D2.2.1 Pseudocode description of routing debug exceptions

[DebugTarget\(\)](#) returns the current debug target Exception level. [DebugTargetFrom\(\)](#) returns the debug target Exception level for the specified Security state.

D2.3 The debug exception enable controls

The enable controls for each debug exception are as follows:

Breakpoint Instruction exceptions

None. Breakpoint Instruction exceptions are always enabled.

Breakpoint exceptions

`DBGDSCRExt.MDBGn`, plus an enable control for each breakpoint, `DBGBCR<n>.E`.

Watchpoint exceptions

`DBGDSCRExt.MDBGn`, plus an enable control for each watchpoint, `DBGWCR<n>.E`.

Vector Catch exceptions

`DBGDSCRExt.MDBGn`.

In addition, for all debug exceptions other than Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level. See [Enabling debug exceptions](#).

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level.

Breakpoint Instruction exceptions are always enabled from the current Exception level.

D2.3.1 Enabling debug exceptions

A debug exception can only be taken if all of the following are true:

- The OS Lock is unlocked.
- `DoubleLockStatus()` == FALSE.
- The debug exception is enabled from the current Privilege level.

[Table D2-3](#) shows when debug exceptions are enabled from the current Privilege level.

Table D2-3 Whether debug exceptions are enabled from the current Privilege level

Current Privilege level	Breakpoint Instruction exceptions	All other debug exceptions
PL2	Enabled	Disabled
PL1	Enabled	Enabled
PL0	Enabled	Enabled

D2.3.2 Pseudocode description of enabling debug exceptions

`AArch32.GenerateDebugExceptions()` determines whether debug exceptions are enabled from the current Exception level. `AArch32.GenerateDebugExceptionsFrom()` determines whether debug exceptions are enabled from the specified Exception level.

D2.4 The effect of powerdown on debug exceptions

[Debug OS Save and Restore sequences](#) describes the *powerdown save routine* and the *restore routine*.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

Debug exceptions other than Breakpoint Instruction exceptions are enabled only if both the OS Lock is unlocked and `DoubleLockStatus() == FALSE`.

Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and the OS Double Lock.

D2.5 Summary of permitted routing and enabling of debug exceptions

Behavior is as follows:

Breakpoint Instruction exceptions

These are always enabled, regardless of the current Privilege level. [Table D2-4](#) shows the routing of these.

Table D2-4 Routing of Breakpoint Instruction exceptions

HDCR.TDE ^a	Target when enabled from:		
	PL0	PL1	PL2
0	Abort mode	Abort mode	Hyp mode
1	Hyp mode	Hyp mode	Hyp mode

a. If the value of [HCR.TGE](#) is 1, [HDCR.TDE](#) is treated as being 1 other than for a direct read of [HDCR](#).

All other debug exceptions

The enabling and permitted routing is controlled by all of the following:

- [HDCR.TDE](#).
- The IMPLEMENTATION DEFINED authentication interface.

[Table D2-5](#) shows the valid combinations of the value of [HDCR.TDE](#). For each combination, the table shows where debug exceptions are enabled from and where they are taken to.

In the table, n/a means not applicable and a dash, -, means that debug exceptions are disabled from that Exception level.

Table D2-5 Breakpoint, Watchpoint, and Vector Catch exceptions

Debug state	Lock ^a	TDE ^b	Target AArch32 mode when enabled from:		
			PL0	PL1	PL2
Yes	X	X	-	-	-
No	TRUE	X	-	-	-
No	FALSE	0	Abort mode	Abort mode	-
No	FALSE	1	Hyp mode	Hyp mode	-

a. The value of `(OSLSR_EL1.OSLK == 1 || DoubleLockStatus())`.

b. If [HCR.TGE](#) is 1, this bit is treated as being 1 other than for a direct read of [HDCR](#). If EL2 is not implemented, behavior is as if TDE is 0.

D2.6 Pseudocode description of debug exceptions

The `AArch32_Abort()` function processes `FaultRecord()`, as described in [Abort exceptions](#), and generates:

- Data Abort exceptions for watchpoints.
- Prefetch Abort exceptions for all other debug exceptions.

D2.7 Breakpoint Instruction exceptions

This section describes Breakpoint Instruction exceptions in an AArch32 translation regime.

It contains the following subsections:

- [About Breakpoint Instruction exceptions.](#)
- [Breakpoint instruction in the T32 and A32 instruction sets.](#)
- [BKPT instructions as the first instruction in an IT block.](#)
- [Exception syndrome information and preferred return address for a BKPT instruction.](#)
- [Pseudocode description of Breakpoint Instruction exceptions.](#)

D2.7.1 About Breakpoint Instruction exceptions

A *breakpoint* is an event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint*.
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint Instruction exceptions, that this section describes, are software breakpoints. [Breakpoint exceptions](#) describes hardware breakpoints.

There is no enable control for Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of the current Exception level.

Note

- EL_D is the Exception level that debug exceptions are targeting. See [Enabling debug exceptions](#).
 - Debuggers using breakpoint instructions must be aware of the rules for concurrent modification and execution of instructions. See [Concurrent modification and execution of instructions](#).
-

D2.7.2 Breakpoint instruction in the T32 and A32 instruction sets

The breakpoint instruction, in both instruction sets, is:

- `BKPT #<immediate>`

For details of the instruction encoding, see [BKPT](#).

D2.7.2.1 About whether the BKPT instruction is conditional

In the T32 instruction set, BKPT instructions are always unconditional.

In the A32 instruction set:

- If the Condition code field is AL, the BKPT instruction is unconditional.
- If the Condition code field is anything other than AL, behavior is CONSTRAINED UNPREDICTABLE, and is one of the following:
 - The instruction is UNDEFINED.
 - The instruction is treated as a NOP instruction.
 - The instruction is executed unconditionally.
 - The instruction is executed conditionally.

D2.7.3 BKPT instructions as the first instruction in an IT block

If the first instruction in an IT block is a T32 BKPT instruction, then in an implementation that supports the ITD control, if ITD field that applies to the current Exception level is:

- | | |
|----------|--|
| 0 | The BKPT instruction generates a Breakpoint Instruction exception. |
| 1 | The combination of IT instruction and BKPT instruction is UNDEFINED. Either the IT instruction or the BKPT instruction generates an Undefined Instruction exception. |

In such an implementation, to ensure consistent behavior when making the first instruction in one or more IT blocks a BKPT instruction, the debugger must replace the IT instruction.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

HSCTLR.ITD Applies to execution at EL2 when EL2.

SCTLR.ITD Applies to execution at EL0 or EL1.

Note

T32 BKPT instructions are always unconditional.

D2.7.4 Exception syndrome information and preferred return address for a BKPT instruction

See the following:

- [Exception syndrome information for a Breakpoint Instruction exception.](#)
- [Preferred return address for a Breakpoint Instruction exception.](#)

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode. The referenced section uses the term more generally, to include exception information reported in the [IFSR](#).

D2.7.4.1 Exception syndrome information for a Breakpoint Instruction exception

The PE takes a Breakpoint Instruction exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp Trap exception, if it is taken to PL2 because either [HCR.TGE](#) or [HDCR.TDE](#) is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

PL1 Abort mode

The PE sets all of the following:

- [DBGDSCRext.MOE](#) to 0b0011, to indicate a Breakpoint Instruction exception.
- [IFSR.FS](#) to the code for a debug, 0b00010.
- The [IFAR](#) with an UNKNOWN value.

PL2 Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, [HSR](#). See [Table D2-6](#).
- Sets [DBGDSCRext.MOE](#) to 0b0011, to indicate a Breakpoint Instruction exception.

- Sets the [HIFAR](#) to an unknown value.

Table D2-6 Information recorded in the HSR

HSR field	Information recorded
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, 0x20.
Instruction Length, IL	The PE sets this to: • 0 for a T32 BKPT instruction. • 1 for an A32 BKPT instruction.
Instruction Specific Syndrome, ISS	ISS[24:10]RES0. ISS[9]External Abort type (EA). The PE sets this to 0. ISS[8:6]RES0. ISS[5:0]Instruction Fault Status Code (IFSC). The PE sets this to the code for a debug exception, 0b100010.

Note
For information about how debug exceptions can be routed to PL2, see [Routing debug exceptions](#).

D2.7.4.2 Preferred return address for a Breakpoint Instruction exception

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like `SVC`.

D2.7.5 Pseudocode description of Breakpoint Instruction exceptions

`AArch32_SoftwareBreakpoint()` generates a Prefetch Abort exception.

D2.8 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of a translation regime.

This section contains the following subsections:

- [About Breakpoint exceptions.](#)
- [Breakpoint types and linking of breakpoints.](#)
- [Execution conditions for which a breakpoint generates Breakpoint exceptions.](#)
- [Breakpoint instruction address comparisons.](#)
- [Breakpoint context comparisons.](#)
- [Using breakpoints.](#)
- [Exception syndrome information and preferred return address for a Breakpoint exception.](#)
- [Pseudocode description of Breakpoint exceptions taken from AArch32 state.](#)

D2.8.1 About Breakpoint exceptions

A *breakpoint* is an event that results from the execution of an instruction, based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint*.
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint exceptions are generated by *Breakpoint debug events*. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in [Breakpoint Instruction exceptions](#).

An implementation can include between 2-16 hardware breakpoints. [DBGDIDR.BRPs](#) shows how many are implemented.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:

- The *Breakpoint Control Register*, [DBGBCR<n>](#). This contains controls for the breakpoint, for example an enable control.
- The *Breakpoint Value Register*, [DBGBVR<n>](#). This holds a value used for breakpoint matching, that is one of:
 - An instruction virtual address.
 - A Context ID.
- The *Breakpoint Extended Value Register*, [DBGBXVR<n>](#), that holds a VMID value used for breakpoint matching.

These registers are numbered, so that:

- [DBGBCR1](#), [DBGBVR1](#), and [DBGBXVR1](#) are for breakpoint number one.
- [DBGBCR2](#), [DBGBVR2](#), and [DBGBXVR2](#) are for breakpoint number two.
- ...
- ...
- [DBGBCR<n>](#), [DBGBVR<n>](#), and [DBGBXVR<n>](#) are for breakpoint number <n>.

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is only generated if both breakpoints match.

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

- The breakpoint is enabled. That is, the breakpoint enable control for it, [DBGBCR<n>.E](#), is 1.
- The conditions specified in the [DBGBCR<n>](#) are met.
- The comparison with the value held in [DBGBVR<n>](#) is successful.

- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

If all of these conditions are met, the breakpoint generates the Breakpoint debug event regardless of the following:

- Whether the instruction passes its Condition code check.
- The instruction type.

If halting is allowed and [EDSCR.HDE](#) is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are

- Enabled, Breakpoint debug events generate Breakpoint exceptions
- Disabled, Breakpoint debug events are ignored.

Note

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions. However, the behavior described also applies if breakpoints are causing entry to Debug state.

[The debug exception enable controls](#) describes the enable controls for Breakpoint debug events.

D2.8.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A *context-aware* breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
 - An instruction address match.
 - An instruction address mismatch.
 - A Context ID match, with the value held in the [CONTEXTIDR](#).
- A breakpoint that is not context-aware. These can only be programmed to generate a Breakpoint exception on an instruction address match or an instruction address mismatch.

[DBGDIDR.CTX_CMPs](#) shows how many of the implemented breakpoints are context-aware breakpoints. At least one implemented breakpoint must be context-aware. The context-aware breakpoints are the highest numbered breakpoints.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match or mismatch is categorized as an *Address breakpoint*. Breakpoints that are programmed to match on anything else are categorized as *Context breakpoints*.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:

- Used in isolation. In this case, the breakpoint is called an *Unlinked breakpoint*.
- Enabled for linking to another breakpoint. In this case, the breakpoint is called a *Linked breakpoint*.

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that only generates a Breakpoint exception if the PE is in a particular context when an instruction address match or mismatch occurs. For example, a debugger might:

1. Program breakpoint number one to be a *Linked Address Match breakpoint*.
2. Program breakpoint number five to be a *Linked Context ID Match breakpoint*.
3. Link these two breakpoints together. A Breakpoint exception is only generated if both the instruction address matches and the Context ID matches.

The *Breakpoint Type* field for a breakpoint, `DBGBCR<n>.BT`, controls the breakpoint type and whether the breakpoint is enabled for linking. If `BT[0]` is 1, the breakpoint is enabled for linking.

Address breakpoints can be programmed to generate Breakpoint exceptions on addresses that are halfword-aligned but not word-aligned. This makes it possible to breakpoint on T32 instructions. See [Specifying the halfword-aligned address that an Address breakpoint matches on](#).

D2.8.2.1 Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, `DBGBCR<n>.LBN`, for the Linked Address breakpoint specifies the particular Linked Context breakpoint that the Linked Address breakpoint links to, and:
 - `DBGBCR<n>.{HMC, PMC}` for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See [Execution conditions for which a breakpoint generates Breakpoint exceptions](#).
 - `DBGBCR<n>.{HMC, PMC}` for the Linked Context breakpoint are ignored.
- Linked Context breakpoint types can only be linked to. The LBN field for Context breakpoints is therefore ignored.
- Linked Address breakpoints cannot link to watchpoints. The LBN field can therefore only specify another breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not context-aware, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See [Other usage constraints for Address breakpoints](#).
- If a Linked Address breakpoint links to an Unlinked Context breakpoint, the Linked Address breakpoint never generates any Breakpoint exceptions.
- Multiple Linked Address breakpoints can link to a single Linked Context breakpoint.

Note

Multiple Linked watchpoints can also link to a single Linked Context breakpoint. [Watchpoint exceptions](#) describes watchpoints.

These rules mean that a single Linked Context breakpoint might be linked to by all, or any combination of, the following:

- Multiple Linked Address Match breakpoints.
- Multiple Linked Address Mismatch breakpoints.
- Multiple Linked watchpoints.

It is also possible that a Linked Context breakpoint might have no breakpoints or watchpoints linked to it.

[Figure D2-1](#) shows an example of permitted breakpoint and watchpoint linking.

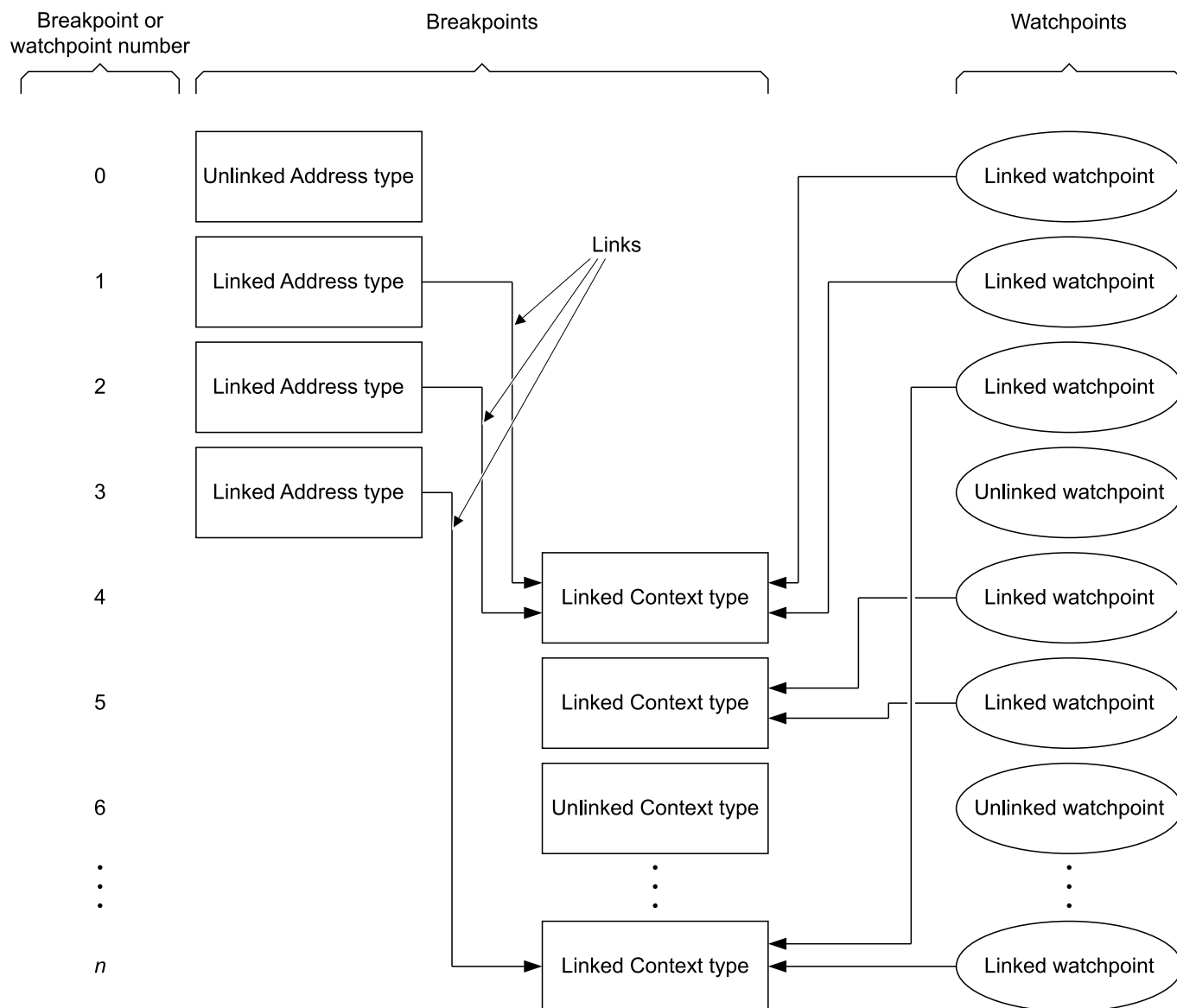


Figure D2-1 The role of linking in Breakpoint and Watchpoint exception generation

In [Figure D2-1](#), each Linked Address breakpoint can only generate a Breakpoint exception if the comparisons made by both it, and the Linked Context breakpoint that it links to, are successful.

D2.8.2.2 Breakpoint types defined by DBGBCRn.BT

The following list provides more detail about each breakpoint type:

0b0000, Unlinked Address Match breakpoint

Generation of a Breakpoint exception depends on both:

- [DBGBCR<n>.{HMC, PMC}](#). These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See [Execution conditions for which a breakpoint generates Breakpoint exceptions](#).
- A successful address match, as described in [Breakpoint instruction address comparisons](#).

[DBGBCR<n>.LBN](#) for this breakpoint is ignored.

0b0001, Linked Address Match breakpoint

Generation of a Breakpoint exception depends on all of the following:

- [DBGBCR<n>.{HMC, PMC}](#) for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See [Execution conditions for which a breakpoint generates Breakpoint exceptions](#).
- A successful address match defined by this breakpoint, as described in [Breakpoint instruction address comparisons](#).
- A successful context match defined by the Context breakpoint with linking enabled that this breakpoint links to.

[DBGBCR<n>.LBN](#) for this breakpoint selects the Context breakpoint with linking enabled that this breakpoint links to.

0b0010, Context ID Match breakpoint

BT = 0b0010 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- [DBGBCR<n>.{HMC, PMC}](#). These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See [Execution conditions for which a breakpoint generates Breakpoint exceptions](#).
- A successful Context ID match, as described in [Breakpoint context comparisons](#).

The value of [DBGBCR<n>.ContextID](#) is compared with the current Context ID.

[CONTEXTIDR](#) holds the current Context ID.

[DBGBCR<n>.{LBN, BAS}](#) for this breakpoint are ignored

0b0011, Context ID Match breakpoint with linking enabled

BT = 0b0011 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, either:

- This breakpoint does not generate any debug exceptions, if no Linked breakpoints or Linked watchpoints link to it.
- Generation of a Breakpoint exception depends on both:
 - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see [Breakpoint instruction address comparisons](#).
 - A successful Context ID match defined by this breakpoint, as described in [Breakpoint context comparisons](#).
- Generation of a Watchpoint exception depends on both:
 - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see [Watchpoint data address comparisons](#).
 - A successful Context ID match defined by this breakpoint, as described in [Breakpoint context comparisons](#).

The value of [DBGBCR<n>.ContextID](#) is compared with the current Context ID.

[CONTEXTIDR](#) holds the current Context ID.

[DBGBCR<n>.{LBN, HMC, BAS, PMC}](#) for this breakpoint are ignored.

0b0100, Unlinked Address Mismatch breakpoint

Generation of a Breakpoint exception depends on both:

- [DBGBCR<n>.{HMC, PMC}](#). These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See [Execution conditions for which a breakpoint generates Breakpoint exceptions](#).
- A successful address mismatch, as described in [Breakpoint instruction address comparisons](#).

[DBGBCR<n>.LBN](#) for this breakpoint is ignored.

0b0101, Linked Address Mismatch breakpoint

Generation of a Breakpoint exception depends on all of the following:

- [DBGBCR<n>.{HMC, PMC}](#). These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See [Execution conditions for which a breakpoint generates Breakpoint exceptions](#).
- A successful address mismatch defined by this breakpoint, as described in [Breakpoint instruction address comparisons](#).
- A successful context match defined by the Linked Context breakpoint that this breakpoint links to.

[DBGBCR<n>.](#)LBN for this breakpoint selects the Linked Context breakpoint that this breakpoint links to.

0b0110, CONTEXTIDR_EL1 Match breakpoint

BT = 0b0110 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- [DBGBCR<n>.{HMC, PMC}](#). These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match defined by this breakpoint, as described in [Breakpoint context comparisons](#).

The Context ID check is made against the value in [CONTEXTIDR](#). The value of [DBGBVR<n>.](#)ContextID is compared with the Context ID value held in [CONTEXTIDR](#).

[DBGBCR<n>.{LBN, BAS}](#) for this breakpoint are ignored.

0b0111, CONTEXTIDR_EL1 Match breakpoint with linking enabled

BT = 0b0111 is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
 - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see [Breakpoint instruction address comparisons](#).
 - A successful Context ID match defined by this breakpoint, as described in [Breakpoint context comparisons](#).
- Generation of a Watchpoint exception depends on both:
 - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see [Watchpoint data address comparisons](#).
 - A successful Context ID match defined by this breakpoint, as described in [Breakpoint context comparisons](#).

The Context ID check is made against the value in [CONTEXTIDR](#). The value of [DBGBVR<n>.](#)ContextID is compared with the Context ID value held in [CONTEXTIDR](#).

[DBGBCR<n>.{LBN, SSC, HMC, BAS, PMC}](#) for this breakpoint are ignored.

Note

See [Reserved DBGBCR<n>.BT values](#) for the behavior of breakpoints programmed with reserved BT values.

D2.8.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it only generates Breakpoint exceptions for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at PL0 in Secure state.

[DBGBCR<n>.{SSC, HMC, PMC}](#) define the execution conditions the breakpoint generates Breakpoint exceptions for, as follows:

Security State Control, SSC

The R-profile AArch32 architecture is always in Non-secure state, and therefore the Security state that the breakpoint targets is always Non-secure.

Higher Mode Control, HMC, and Privileged Mode Control, PMC

HMC and PMC together control which AArch32 modes the breakpoint generates Breakpoint exceptions in.

[Table D2-7](#) shows the valid combinations of the values of HMC, SSC, and PMC, and for each combination shows which Privilege levels breakpoints generate Breakpoint exceptions in.

In the table:

- Y** Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row can generate Breakpoint exceptions in AArch32 modes at that Privilege level.
- Means that a breakpoint programmed with the values of HMC, SSC and PMC shown in that row cannot generate Breakpoint exceptions in AArch32 modes at that Privilege level.

A combination in [Table D2-7](#) might be reserved if an Exception level or Security state is not implemented. For information about which combinations of HMC, SSC and PMC are reserved if an Exception level or Security state are not implemented, see [Reserved DBGBCR<n>.{SSC, HMC, PMC} values](#).

Table D2-7 Summary of breakpoint HMC, SSC, and PMC encodings

HMC	SSC	PMC	PL2% ^a	PL1	PL0
0b0	0b00	0b00	-	Y% ^b	Y
0b0	0b00	0b01	-	Y	-
0b0	0b00	0b10	-	-	Y
0b0	0b00	0b11	-	Y	Y
0b1	0b00	0b01	Y	Y	Y
0b1	0b00	0b11	Y	Y	Y
0b1	0b11	0b00	Y	-	-

a. Debug exceptions are not generated at PL2. This means that these combinations of HMC and PMC are only relevant if breakpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC and PMC that generate Breakpoint exceptions at PL2.

b. Only in User and System modes.

All combinations of HMC, SSC, and PMC that this table does not show are reserved. See [Table D2-10](#).

D2.8.4 Breakpoint instruction address comparisons

Address comparisons are made for each instruction in the program flow. The following subsections describe the criteria for a successful address comparison, for:

- [Address Match breakpoints.](#)
- [Address Mismatch breakpoints.](#)

D2.8.4.1 Address Match breakpoints

An address match comparison is successful if both:

- Bits [31:2] of the current instruction virtual address are equal to [DBGVCR<n>](#)[31:2].
 - The word or halfword selected by [DBGVCR<n>](#).BAS matches. That is, either:
 - [DBGVCR<n>](#).BAS is programmed with 0b0011 or 0b1111, and the instruction is at a word-aligned address.
 - [DBGVCR<n>](#).BAS is programmed with 0b1100, and the instruction is not at a word-aligned address.
- See [Specifying the halfword-aligned address that an Address breakpoint matches on.](#)

Note

[DBGVCR<n>](#)[1:0] are RES0 and are ignored.

D2.8.4.2 Address Mismatch breakpoints

An address mismatch comparison is successful if either:

- Bits [31:2] of the current instruction virtual address are not equal to [DBGVCR<n>](#)[31:2].
 - The word or halfword selected by [DBGVCR<n>](#).BAS does not match. That is, either:
 - [DBGVCR<n>](#).BAS is programmed with 0b0011 or 0b1111, and the instruction is not at a word-aligned address.
 - [DBGVCR<n>](#).BAS is programmed with 0b1100, and the instruction is at a word-aligned address.
- See [Specifying the halfword-aligned address that an Address breakpoint matches on.](#)

Note

- [DBGVCR<n>](#)[1:0] are RES0 and are ignored.
 - Address Mismatch breakpoints can be used to single-step through code. See [Using an Address Mismatch breakpoint to single-step an instruction.](#)
-

D2.8.4.3 Specifying the halfword-aligned address that an Address breakpoint matches on

For an Address breakpoint, a debugger can use the *Byte Address Selection* field, [DBGVCR<n>](#).BAS, so that the address comparison is successful on one of:

- The whole word starting at address [DBGVCR<n>](#)[31:2]:00.
- The halfword starting at address [DBGVCR<n>](#)[31:2]:00.
- The halfword starting at address (([DBGVCR<n>](#)[31:2]:00) + 2).

Note

The address programmed into the [DBGVCR<n>](#) must be word-aligned.

[DBGVCR<n>](#).BAS can be used in both Address Match breakpoints and Address Mismatch breakpoints, as follows:

- For an Address Match breakpoint, [DBGVCR<n>](#).BAS selects which halfword-aligned address the breakpoint must generate a Breakpoint exception for. This means that an address comparison is successful only if both of the following match:
 - The instruction address held in bits [31:2] of the [DBGVCR<n>](#).

— The halfword defined by the BAS field.

That is, a successful address comparison = `DBGBVR<n>[31:2]` match AND BAS match.

- For an Address Mismatch breakpoint, `DBGBCR<n>`.BAS selects which halfword-aligned address the breakpoint must not generate a Breakpoint exception for. This means that an address comparison is successful if either or both of the following do not match:

— The instruction address held in bits [31:2] of the `DBGBVR<n>`.

— The halfword defined by the BAS field.

That is, a successful address comparison = NOT (`DBGBVR<n>[31:2]` match AND BAS match).

The following subsections show the supported BAS values:

- [Using the BAS field in Address Match breakpoints.](#)
- [Using the BAS field in Address Mismatch breakpoints.](#)

For Context breakpoints, `DBGBCR<n>`.BAS is RES1 and is ignored.

D2.8.4.3.1 Using the BAS field in Address Match breakpoints

The supported BAS values are:

0b0000 This value is reserved. Behavior is a CONSTRAINED UNPREDICTABLE choice of:

- The breakpoint is disabled.
- The breakpoint behaves as if BAS is 0b0011, 0b1100, or 0b1111.

0b0011 The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals `DBGBVR<n>[31:2]`.
- Bits [1:0] of the address are 0b00.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for T32 instructions.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address ((`DBGBVR<n>[31:2]`:00) - 2).

0b1100 The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals `DBGBVR<n>[31:2]`.
- Bits [1:0] of the address are 0b10.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for both of the following:

- 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
- 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

It is CONSTRAINED UNPREDICTABLE whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction starting at a word-aligned address.

0b1111 The breakpoint generates a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals `DBGBVR<n>[31:2]`.
- Bits [1:0] of the address are `0b00`.

This means that breakpoints programmed with this BAS value generate Breakpoint exceptions for all of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for A32 instructions.

It is **CONSTRAINED UNPREDICTABLE** whether a breakpoint programmed with this BAS value generates a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address $((\text{DBGBVR}\langle n \rangle[31:2]:00) - 2)$.

It is **CONSTRAINED UNPREDICTABLE** whether a breakpoint programmed with this BAS value generates a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address $((\text{DBGBVR}\langle n \rangle[31:2]:00) + 2)$.

All other BAS values are reserved. For these reserved other values, `DBGBCR<n>.BAS[3,1]` ignore writes and read the same values as `DBGBCR<n>[2,0]` respectively. This means that the smallest instruction size a debugger can program breakpoints to match on is a halfword.

Figure D2-2 shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 2 shows that a breakpoint with a BAS value of `0b1100` generates Breakpoint exceptions for 16-bit T32 instructions starting at the halfword-aligned address $((\text{DBGBVR}\langle n \rangle[31:2]:00) + 2)$.
- Row 6 shows that a breakpoint with a BAS value of either `0b0011` or `0b1111` generates Breakpoint exceptions for A32 instructions. A32 instructions are always at word-aligned addresses.

In the figure:

Yes	Means that the breakpoint generates a Breakpoint exception.
No	Means that the breakpoint does not generate a Breakpoint exception.
UNP	Means that it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints .

	Instruction set	Size	Location of instruction ^a								BAS[3:0]		
			-2	-1	0	+1	+2	+3	+4	+5	0b0011	0b1100	0b1111
Row 1	T32	16-bit									Yes	No	Yes
Row 2		16-bit									No	Yes	UNP
Row 3	T32	32-bit									UNP	No	UNP
Row 4		32-bit									Yes	UNP	Yes
Row 5		32-bit									No	Yes	UNP
Row 6	A32	32-bit									Yes	UNP	Yes

a. 0 means the word-aligned address held in the DBGBVRn. The other locations are as follows:

- -2 means ((DBGBVRn[31:2]:00) - 2).
- -1 means ((DBGBVRn[31:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.

Figure D2-2 Summary of BAS field meanings for Address Match breakpoints

D2.8.4.3.2 Using the BAS field in Address Mismatch breakpoints

An Address Mismatch breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed to match.

The supported BAS values are:

- 0b0000 The breakpoint ignores the address held in the [DBGBVR<n>](#) and generates Breakpoint exceptions for all instruction addresses.
- 0b0011 The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:
- Bits [31:2] of the address equals [DBGBVR<n>](#)[31:2].
 - Bits [1:0] of the address are 0b00.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for T32 instructions.

It is **CONSTRAINED UNPREDICTABLE** whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address (([DBGBVR<n>](#)[31:2]:00) - 2).

- 0b1100 The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:
- Bits [31:2] equals [DBGBVR<n>](#)[31:2].
 - Bits [1:0] of the address are 0b10.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for either of the following:

- 32-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.
- 16-bit T32 instructions at addresses that are halfword-aligned but not word-aligned.

It is **CONSTRAINED UNPREDICTABLE** whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 or A32 instruction at a word-aligned address.

0b1111 The breakpoint does not generate a Breakpoint exception if an instruction with an address described as follows is committed for execution:

- Bits [31:2] of the address equals [DBGBVR<n>](#)[31:2].
- Bits [1:0] of the address are 0b00.

This means that breakpoints programmed with this BAS value do not generate Breakpoint exceptions for any of the following:

- 32-bit T32 instructions at word-aligned addresses.
- 16-bit T32 instructions at word-aligned addresses.
- A32 instructions. These are always at word-aligned addresses.

However, Arm recommends that a debugger uses this BAS value only for A32 instructions.

It is **CONSTRAINED UNPREDICTABLE** whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on the second halfword of a 32-bit T32 instruction starting at the halfword-aligned address (([DBGBVR<n>](#)[31:2]:00) - 2).

It is **CONSTRAINED UNPREDICTABLE** whether a breakpoint programmed with this BAS value does not generate a Breakpoint exception on a 32-bit T32 instruction or a 16-bit T32 instruction at the halfword-aligned address (([DBGBVR<n>](#)[31:2]:00) + 2).

All other BAS values are reserved. For these reserved other values, [DBGBCR<n>](#).BAS[3,1] ignore writes and read the same values as [DBGBCR<n>](#)[2,0] respectively. This means that the smallest instruction size that a breakpoint can never generate a Breakpoint exception for is a halfword.

[Figure D2-3](#) shows a summary of when breakpoints programmed with particular BAS values generate Breakpoint exceptions.

The figure contains four parts:

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example:

- Row 1 shows that a breakpoint with a BAS value of 0b1100 generates Breakpoint exceptions for 16-bit T32 instructions starting at the word-aligned address held in the [DBGBVR<n>](#).
- Row 5 shows that a breakpoint with a BAS value of 0b0011 generates Breakpoint exceptions for 32-bit T32 instructions starting at the halfword-aligned address immediately after the word aligned address held in the [DBGBVR<n>](#).

In the figure:

Yes	Means that the breakpoint does generate a Breakpoint exception.
No	Means that the breakpoint does not generate a Breakpoint exception.
UNP	Means that it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints .

			Location of instruction ^a							BAS[3:0]				
	Instruction set	Size	-2	-1	0	+1	+2	+3	+4	+5	0b0000	0b0011	0b1100	0b1111
Row 1	T32	16-bit									Yes	No	Yes	No
Row 2		16-bit									Yes	Yes	No	UNP
Row 3	T32	32-bit									Yes	UNP	Yes	UNP
Row 4		32-bit									Yes	No	UNP	No
Row 5		32-bit									Yes	Yes	No	UNP
Row 6	A32	32-bit									Yes	No	UNP	No

a. 0 means the word-aligned address held in the DBGVRn. The other locations are as follows:

- -2 means ((DBGVRn[31:2]:00) - 2).
- -1 means ((DBGVRn[31:2]:00) - 1).
- ...
- ...
- +5 means ((DBGVRn[31:2]:00) + 5).

The solid areas show the location of the instruction.

Figure D2-3 Summary of BAS field meanings for Address Mismatch breakpoints

D2.8.5 Breakpoint context comparisons

The breakpoint type defined by **DBGBCR<n>.BT** determines what context comparison is required, if any. [Table D2-8](#) shows the BT values that require a comparison, and the match required for the comparison to be successful.

Table D2-8 Breakpoint Context ID and VMID comparison tests

DBGBCR<n>.BT	Test required for successful context comparison
0b001x	CONTEXTIDR must match the DBGVR<n>.ContextID value.
0b011x	Reserved.
0b100x	VSCTLR.VMID must match the DBGXVR<n>.VMID value.
0b101x	CONTEXTIDR , must match the DBGVR<n>.ContextID value, and VSCTLR.VMID must match the DBGXVR<n>.VMID value.
0b111x	Reserved.

No context comparison is required for other valid **DBGBCR<n>.BT** values.

Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of **CONTEXTIDR** and the PE is executing at EL2.

Note

- For all Context breakpoints, **DBGBCR<n>.BAS** is RES1 and is ignored.
- For Linked Context breakpoints, **DBGBCR<n>.{LBN, SSC, HMC, PMC}** are RES0 and are ignored.

D2.8.6 Using breakpoints

This section contains the following:

- [Using an Address Mismatch breakpoint to single-step an instruction.](#)

- [ITD control effects on address breakpoints on the first instruction in an IT block.](#)
- [Breakpoint usage constraints.](#)

D2.8.6.1 Using an Address Mismatch breakpoint to single-step an instruction

In execution conditions that an Address Mismatch breakpoint matches, defined by `DBGBCR<n>.{LBN, SSC, PMC}`, the breakpoint generates Breakpoint exceptions for all instructions committed for execution, except the instruction whose address the breakpoint is programmed with. [Figure D2-4](#) shows an example of Address Mismatch breakpoint operation, for an Address Mismatch breakpoint programmed with address `0x1014`.

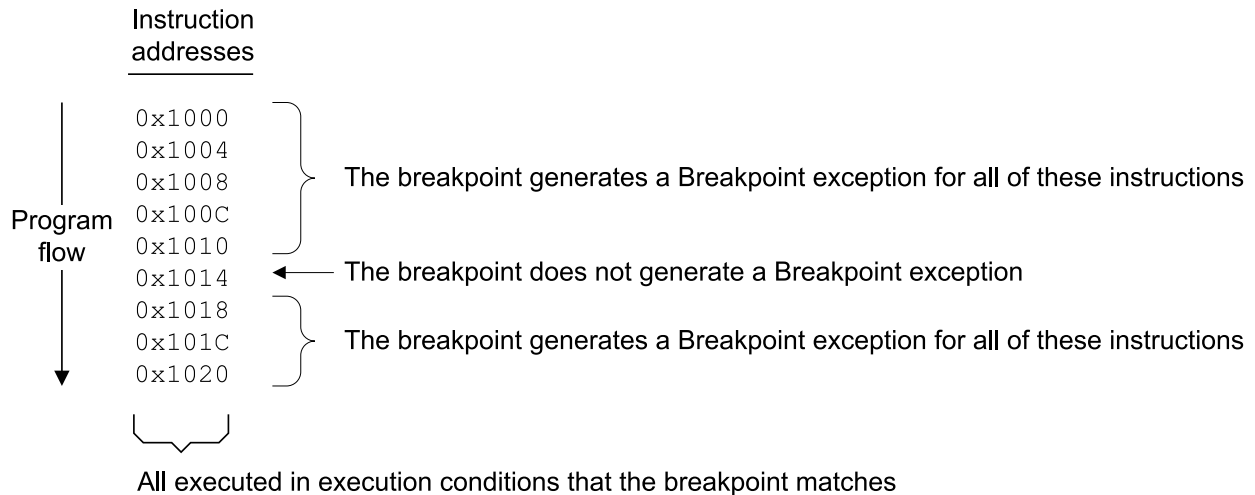


Figure D2-4 Operation of an Address Mismatch breakpoint

This means that an Address Mismatch breakpoint can be used to single-step an instruction.

In the example shown in [Figure D2-4](#):

- If the target of a branch is an instruction other than the instruction at address `0x1014`, the breakpoint generates a Breakpoint exception when the instruction is committed for execution.
- If the target of a branch is the instruction at address `0x1014`, the PE executes the instruction at `0x1014` and the breakpoint does not generate a Breakpoint exception until the instruction at address `0x1018` is committed for execution. The instruction at address `0x1014` is therefore single-stepped.

However, if the instruction at `0x1014` generates a synchronous exception, or if the PE takes an asynchronous exception while the instruction is being stepped, the breakpoint is evaluated again after taking the exception. This means that behavior is as follows:

- If the exception handler executes in execution conditions that the breakpoint matches, the breakpoint generates a Breakpoint exception for the exception vector, because the exception vector is not address `0x1014`. This means that software execution steps into the exception.
- If the exception handler executes in execution conditions that the breakpoint does not match, the breakpoint does not generate any Breakpoint exceptions after the PE has taken the exception, until the exception handler completes and executes an exception return instruction. The effect is to step over the exception. Whether the instruction is stepped again depends on whether the target of the exception return instruction is the instruction at `0x1014` or the instruction at `0x1018`.

If the instruction at `0x1014` is single-stepped and branches to itself, it is **CONSTRAINED UNPREDICTABLE** whether the breakpoint generates a Breakpoint exception after the PE has executed the branch.

This means that an instruction is only single-stepped if it is the target of a branch instruction and its address matches the address the breakpoint is programmed for. In the example shown in [Figure D2-4](#), this is `0x1014`.

Usually this branch instruction is an exception return instruction that changes PE mode, branching from a PE mode in which the breakpoint does not generate a Breakpoint exception. A branch instruction that does not change PE mode would itself generate a Breakpoint exception. However, it might be a branch-to-self instruction as described above.

Note

The example shown in [Figure D2-4](#) assumes an A32 instruction.
The same behavior applies for both 32-bit and 16-bit T32 instructions.

D2.8.6.2 ITD control effects on address breakpoints on the first instruction in an IT block

In an implementation that supports the ITD control, if the value of the ITD field that applies to the current Exception level is 1, all of the following are true:

- An `IT` instruction can only be used to apply to one 16-bit T32 instruction.
- Only certain combinations of an `IT` instruction and second single 16-bit T32 instruction are permitted.
- For a permitted combination, it is IMPLEMENTATION DEFINED whether the implementation treats the combination as:
 - A pair of 16-bit instructions.
 - One 32-bit instruction.

If the implementation treats the combination as one 32-bit instruction, then as described in [Other usage constraints for Address breakpoints](#), an Address breakpoint might not generate a Breakpoint exception for an address match only on the second halfword of the instruction.

For this reason, if the ITD bit associated with the current Exception level is 1, Arm recommends that a debugger that wants to program a breakpoint to match on the second T32 instruction programs it to match on the `IT` instruction instead.

However, if returning from an exception whose preferred return address is the address of the second T32 instruction, then because the debugger is aware that the implementation has treated the combination as a pair of 16-bit instructions, the debugger is permitted to program the breakpoint to match on the second T32 instruction.

The ITD control fields are:

HSCTLR.ITD Applies to execution at EL2.

SCTLR.ITD Applies to execution at EL0 or EL1.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0, and therefore the information in this section does not apply to such an implementation.

Note

Programming the breakpoint to match on the second T32 instruction might be necessary when using an Address Mismatch breakpoint for single stepping.

D2.8.6.3 Breakpoint usage constraints

See the following sections:

- [Reserved DBGBCR<n>.BT values.](#)
- [Reserved DBGBCR<n>.{SSC, HMC, PMC} values.](#)
- [Reserved DBGBCR<n>.BAS values.](#)
- [Reserved DBGBCR<n>.LBN values.](#)
- [Other usage constraints for Address breakpoints.](#)
- [Other usage constraints for Context breakpoints.](#)

D2.8.6.3.1 Reserved DBGBCR<n>.BT values

[Table D2-9](#) shows when particular `DBGBCR<n>.BT` values are reserved.

Table D2-9 Reserved BT values

BT value	Breakpoint type	Reserved
0b001x	Context ID Match	If the breakpoint is not context-aware
0b010x	Address Mismatch	If EDSCR.HDE is 1 and halting is allowed
0b011x	•	Reserved
0b100x	Context ID Match and VMID Match	If the breakpoint is not context-aware
0b101x	•	Reserved
0b11xx	•	Reserved

If a breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
 - Disabled.
 - Programmed with a BT value that is not reserved, other than for a direct or external read of [DBGBCR<n>](#).
- For a direct or external read of [DBGBCR<n>](#), if the reserved BT value:
 - Has no function for any execution conditions, the value read back is UNKNOWN.
 - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the BT value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved BT values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

D2.8.6.3.2 Reserved [DBGBCR<n>](#).{SSC, HMC, PMC} values

[Table D2-10](#) shows when particular combinations of [DBGBCR<n>](#).{SSC, HMC, PMC} are reserved.

Table D2-10 Reserved HMC and PMC combinations

HMC, and PMC combination	Reserved
Combinations not included in Table D2-7 .	Always.

For all breakpoints except Linked Context breakpoints, if a breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
 - The breakpoint must behave as if it is disabled.
 - A direct or external read of [DBGBCR<n>](#).{SSC, HMC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.
- If the reserved combination does not have a function for other execution conditions:
 - It must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
 - A direct or external read of [DBGBCR<n>](#).{SSC, HMC, PMC} returns UNKNOWN values.

If the breakpoint is a Linked Context breakpoint, then:

- The values of HMC, SSC, and PMC are ignored.

- A direct or external read of `DBGBCR<n>.{SSC, HMC, PMC}` returns UNKNOWN values

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

D2.8.6.3.3 Reserved `DBGBCR<n>.BAS` values

For all Context breakpoints

`DBGBCR<n>.BAS` is RES1 and is ignored.

For all Address breakpoints

The supported values of the BAS field for the Address Match and Address Mismatch breakpoints are shown in [Specifying the halfword-aligned address that an Address breakpoint matches on](#).

If a breakpoint is programmed with a reserved BAS value:

- The breakpoint must behave as if it is either:
 - Disabled.
 - Programmed with a BAS value that is not reserved, other than for a direct or external read of `DBGBCR<n>`.
- A direct or external read of `DBGBCR<n>.BAS` returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

D2.8.6.3.4 Reserved `DBGBCR<n>.LBN` values

For all Context breakpoints

`DBGBCR<n>.LBN` reads UNKNOWN and its value is ignored.

For Linked Address breakpoints

A Linked Address breakpoint must link to a context-aware breakpoint. For a Linked Address breakpoint, any `DBGBCR<n>.LBN` value that is not for a context-aware breakpoint is reserved.

If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of `DBGBCR<n>.LBN` return an unknown value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked Address breakpoint that links to a breakpoint that is implemented and that is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked Address breakpoints

`DBGBCR<n>.LBN` reads UNKNOWN and its value is ignored.

D2.8.6.3.5 Other usage constraints for Address breakpoints

For all Address breakpoints

- `DBGBVR<n>[1:0]` are RES0 and are ignored.
- The `DBGBXVR<n>` is ignored.

For Address Match breakpoints

- For 32-bit instructions, if a breakpoint matches on the address of the second halfword but not the address of the first halfword, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception.

- If [DBGBCR<n>](#).BAS is 0b1111, it is CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception for a T32 instruction starting at address $((\text{DBGBVR}<n>[31:2]:00) + 2)$. For T32 instructions, Arm recommends that the debugger programs the BAS field with either 0b0011 or 0b1100.

For Address Mismatch breakpoints

The constraints are the same as those described in [For Address Match breakpoints](#), except that if two Address Mismatch breakpoints are programmed to match in the same Exception level and Security state, it is CONSTRAINED UNPREDICTABLE whether or not the instruction is stepped or a Breakpoint debug event is generated.

D2.8.6.3.6 Other usage constraints for Context breakpoints

For all Context breakpoints

Any bits of [DBGBVR<n>](#) and [DBGBXVR<n>](#) that are not used to specify Context ID or VMID are RES0 and are ignored.

Note

This means that for Context ID Match breakpoints, the [DBGBXVR<n>](#) is RES0 and is ignored, and for VMID Match breakpoints, the [DBGBVR<n>](#) is RES0 and is ignored.

For Linked Context breakpoints

If no Linked Address breakpoints or Linked Watchpoints link to a Linked Context breakpoint, the Linked Context breakpoint does not generate any Breakpoint exceptions.

D2.8.7 Exception syndrome information and preferred return address for a Breakpoint exception

See the following:

- [Exception syndrome information for a Breakpoint exception](#).
- [Preferred return address for a Breakpoint exception](#).

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode. The referenced section uses the term more generally, to include exception information reported in the [IFSR](#).

D2.8.7.1 Exception syndrome information for a Breakpoint exception

The PE takes a Breakpoint exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because [HCR](#).TGE or [HDCR](#).TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

Abort mode

The PE sets all of the following:

- [DBGDSCRExt](#).MOE to 0b0001, to indicate a Breakpoint exception.
- [IFSR](#).FS to the code for a debug exception, 0b00010.
- The [IFAR](#) with an UNKNOWN value.

Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, [HSR](#). See [Table D2-11](#).
- Sets [DBGDSCRext.MOE](#) to 0b0001, to indicate a Breakpoint exception.
- Sets the [HIFAR](#) to an unknown value.

Table D2-11 Information recorded in the HSR

HSR field	Information recorded	
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, 0x20.	
Instruction Length, IL	The PE sets this to 1.	
Instruction Specific Syndrome, ISS	ISS[24:10]	RES0.
	ISS[9]	External Abort type (EA). The PE sets this to 0.
	ISS[8:6]	RES0.
	ISS[5:0]	Instruction Fault Status Code (IFSC). The PE sets this to the code for a debug exception, 0b100010.

———— **Note** ————
For information about how debug exceptions can be routed to PL2, see [Routing debug exceptions](#).

D2.8.7.2 Preferred return address for a Breakpoint exception

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.8.8 Pseudocode description of Breakpoint exceptions taken from AArch32 state

[AArch32_BreakpointValueMatch\(\)](#) returns a pair of results:

- A result for Address Match and Context breakpoints.
- A result for Address Mismatch breakpoints.

[AArch32_StateMatch\(\)](#) tests the values in [DBGBCR<n>.{SSC, HMC, PMC}](#) and, if the breakpoint links to a Linked Context breakpoint, also tests the Linked Context breakpoint.

[AArch32_BreakpointMatch\(\)](#) tests a committed instruction against all breakpoints.

[AArch32_CheckBreakpoint\(\)](#) generates a [FaultRecord](#). A Breakpoint exception is taken if all of the following are true:

- [DBGDSCRext.MDBGen](#) is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See [Enabling debug exceptions](#).
- All of the conditions required for Breakpoint exception generation are met. See [About Breakpoint exceptions](#).

———— **Note** ————
[AArch32_CheckBreakpoint\(\)](#) might halt the PE and cause it to enter Debug state. External debug uses Debug state.

The [AArch32_Abort\(\)](#) function processes the [FaultRecord](#) object returned by [AArch32_CheckBreakpoint\(\)](#), as described in [Abort exceptions](#). When a Breakpoint exception is taken to AArch32 state, the [AArch32_Abort\(\)](#) function generates a Prefetch Abort exception.

D2.9 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch32 translation regime.

This section contains the following subsections:

- [About Watchpoint exceptions.](#)
- [Watchpoint types and linking of watchpoints.](#)
- [Execution conditions for which a watchpoint generates Watchpoint exceptions.](#)
- [Watchpoint data address comparisons.](#)
- [Watchpoint behavior for certain instruction classes.](#)
- [Watchpoint usage constraints.](#)
- [Exception syndrome information and preferred return address.](#)
- [Pseudocode description of Watchpoint exceptions taken from AArch32 state.](#)

D2.9.1 About Watchpoint exceptions

A *watchpoint* is an event that results from the execution of an instruction, based on a data address. Watchpoints are also known as *data breakpoints*.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a *Watchpoint debug event* on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, [DBGDIDR](#).WRPs shows how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The *Watchpoint Control Register*, [DBGWCR<n>](#). This holds control information for the watchpoint, for example an enable control.
- The *Watchpoint Value Register*, [DBGWVR<n>](#). This holds the data virtual address used for watchpoint matching.

The registers are numbered, so that:

- DBGWCR1 and DBGWVR1 are for watchpoint number one.
- DBGWCR2 and DBGWVR2 are for watchpoint number two.
- ...
- ...
- DBGWCRn and DBGWVRn are for watchpoint number n.

A watchpoint can:

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a *Linked Context breakpoint*, so that a Watchpoint debug event is only generated if the PE is in a particular context when the address match occurs.

A single watchpoint can be programmed to match on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, `DBGWCR<n>.BAS`, to select the bytes. See [Programming a watchpoint with eight bytes or fewer](#).
- Eight bytes to 2GB, provided that both of the following are true:
 - The number of bytes is a power-of-two.
 - The range starts at an address that is aligned to the range size.
 A debugger uses the *MASK* field, `DBGWCR<n>.MASK`, to program a watchpoint with eight bytes to 2GB. See [Programming a watchpoint with eight or more bytes](#).

A debugger must use either the *BAS* field or the *MASK* field. If it uses both, whether the watchpoint generates Watchpoint exceptions is *CONSTRAINED UNPREDICTABLE*. See [Programming dependencies of the BAS and MASK fields](#).

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The conditions specified in the `DBGWCR<n>` are met. This includes all of the following:
 - The watchpoint is enabled. See `DBGWCR<n>.E`.
 - The type of the access. See `DBGWCR<n>.LST`.
 - The Exception level and privilege of the access. See [Execution conditions for which a watchpoint generates Watchpoint exceptions](#).
 - If the watchpoint links to a breakpoint with linking enabled, the comparison or comparisons made by the breakpoint. See [Rules for linking watchpoints](#).
 Together, these define the *context* of the data access that the watchpoint matches.
- The comparison with the address held in the `DBGWVR<n>` is successful. See [Watchpoint data address comparisons](#).
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its Condition code check.

If multiple watchpoints can generate a Watchpoint debug event for an access, then for all watchpoints that are enabled and match the context of the data access, the Boolean equality comparison results of all the watchpoints are OR'd together to generate a result.

If halting is allowed and `EDSCR.HDE` is 1, Watchpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:

- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

Note

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions. However, the behavior described also applies if watchpoints are causing entry to Debug state.

[The debug exception enable controls](#) describes the enable controls for Watchpoint debug events.

D2.9.2 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that it is either:

- Used in isolation. In this case, the watchpoint is called an *Unlinked watchpoint*.
- Enabled for linking to a Linked Context breakpoint. In this case, the watchpoint is called a *Linked watchpoint*.

When a Linked watchpoint links to a Linked Context breakpoint, the Linked watchpoint only generates a Watchpoint exception if the PE is in a particular context when the data address match occurs. For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked Context Match breakpoint*.

3. Link the watchpoint and the breakpoint together. A Watchpoint exception is only generated if both the data address matches and the Context matches.

The *Watchpoint Type* field for a watchpoint, `DBGWCR<n>.WT`, controls whether the watchpoint is enabled for linking. If `DBGWCR<n>.WT` is 1, the watchpoint is enabled for linking.

D2.9.2.1 Rules for linking watchpoints

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can link to any type of Linked Context breakpoint. The *Linked Breakpoint Number* field, `DBGWCR<n>.LBN`, for the Linked watchpoint specifies the particular Linked Context breakpoint that the Linked watchpoint links to, and:
 - `DBGWCR<n>.WT`. {SSC, HMC, PAC} for the Linked watchpoint define the execution conditions that the watchpoint generates Watchpoint exceptions for. See [Execution conditions for which a watchpoint generates Watchpoint exceptions](#).
 - `DBGBCR<n>.{SSC, HMC, PMC}` for the Linked Context breakpoint are ignored.
- A Linked watchpoint cannot link to another watchpoint. The LBN field can therefore only specify a breakpoint.
- If a Linked watchpoint links to a breakpoint that is not context-aware, the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE. See [Watchpoint usage constraints](#).
- If a Linked watchpoint links to an Unlinked Context breakpoint, the Linked watchpoint never generates any Watchpoint exceptions.
- Multiple Linked watchpoints can link to a single Linked Context breakpoint.

Note

Multiple Address breakpoints can also link to a single Linked Context breakpoint. [Breakpoint exceptions](#) describes breakpoints.

[Figure D2-1](#) shows an example of permitted watchpoint linking.

D2.9.3 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it only generates Watchpoint exceptions for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only when the PE is executing at EL2.

`DBGWCR<n>.{SSC, HMC, PAC}` define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

Security State Control, SSC

Controls whether the watchpoint generates Watchpoint exceptions only in Secure state, only in Non-secure state, or in both Security states.

Note

This is determined by the Security state of the PE, not from the NS attribute returned by the translation of the virtual address on which the watchpoint is set.

Higher Mode Control, HMC, and Privileged Access Control, PAC

HMC and PAC together control which Exception levels and Privilege levels the watchpoint generates Watchpoint exceptions in.

The PAC control relates to the privilege of the memory access and Privilege level at which the access was made.

Example D2-1

PAC condition for generating a Watchpoint exception

This means that, if the PE executes a Load unprivileged or Store unprivileged instruction at PL1, the resulting data access triggers a watchpoint only if both:

- PAC is programmed to a value that generates watchpoints on PL0 accesses.
- All other conditions for generating the watchpoint are met.

Example A32/T32 Load unprivileged and Store unprivileged instructions are `LDRT` and `STRT`.

Table D2-12 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Privilege levels watchpoints generate Watchpoint exceptions in.

In the table:

Y or -	Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:
Y	Can generate Watchpoint exceptions at that Privilege level.
-	Cannot generate Watchpoint exceptions at that Privilege level.
Res	Means that the combination of HMC, SSC, and PAC is reserved. See Reserved DBGWCR<n>.{SSC, HMC, PAC} values .

Table D2-12 Summary of watchpoint HMC, SSC, and PAC encodings

HMC	SSC	PAC	PL2 ^a	PL1	Unprivileged
0b0	0b00	0b01	-	Y	-
0b0	0b00	0b10	-	-	Y
0b0	0b00	0b11	-	Y	Y
0b1	0b00	0b01	Y	Y	-
0b1	0b00	0b11	Y	Y	Y
0b1	0b11	0b00	Y	-	-

a. Debug exceptions are not generated at PL2 using AArch32. This means that these combinations of HMC, SSC, and PAC are only relevant if watchpoints cause entry to Debug state. Self-hosted debuggers must avoid combinations of HMC, SSC, and PAC that generate Watchpoint exceptions at PL2 using AArch32.

All combinations of HMC, SSC, and PAC that this table does not show are reserved. See [Reserved DBGWCR<n>.{SSC, HMC, PAC} values](#).

D2.9.4 Watchpoint data address comparisons

An address comparison is successful if bits [31:2] of the current data virtual address are equal to `DBGWVR<n>[31:2]`, taking into account all of the following:

- The size of the access. See [Size of the data access](#).
- The bytes selected by `DBGWVR<n>.BAS`. See [Programming a watchpoint with eight bytes or fewer](#).
- Any address ranges indicated by `DBGWVR<n>.MASK`. See [Programming a watchpoint with eight or more bytes](#).

Note

`DBGWVR<n>[1:0]` are `RES0` and are ignored.

D2.9.4.1 Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each access must be taken into account. See [Example D2-2](#).

Example D2-2

1. A debugger programs a watchpoint to generate Watchpoint exceptions only when the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint must generate a Watchpoint exception.

The size of data accesses initiated by `DCIMVAC` instructions is an IMPLEMENTATION DEFINED size that is both:

- From the inclusive range between:
 - The size that `CTR.DminLine` defines.
 - 2KB.
- A power-of-two.

The lowest address accessed by a `DCIMVAC` instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.

The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also, [Watchpoint behavior on accesses by DCIMVAC instructions](#).

D2.9.4.2 Programming a watchpoint with eight bytes or fewer

The Byte Address Select field, `DBGWCR<n>.BAS`, selects which bytes in the doubleword starting at the address contained in the `DBGWVR<n>` the watchpoint generates Watchpoint exceptions for.

If the address programmed into the `DBGWVR<n>` is:

- Doubleword-aligned:
 - All eight bits of `DBGWCR<n>.BAS` are used, and the descriptions given in [Table D2-13](#) apply.
- Word-aligned but not doubleword-aligned:
 - Only `DBGWCR<n>.BAS[3:0]` are used, and the descriptions given in [Table D2-14](#) apply. In this case, `DBGWCR<n>.BAS[7:4]` are RES0.

Table D2-13 Supported BAS values when the DBGWVRn address alignment is doubleword

BAS value	Description
<code>BAS[0] == 1</code>	Byte at address <code>DBGWVR<n>[31:3]:000</code> selected.
<code>BAS[1] == 1</code>	Byte at address <code>DBGWVR<n>[31:3]:001</code> selected.
<code>BAS[2] == 1</code>	Byte at address <code>DBGWVR<n>[31:3]:010</code> selected.
<code>BAS[3] == 1</code>	Byte at address <code>DBGWVR<n>[31:3]:011</code> selected.
<code>BAS[4] == 1</code>	Byte at address <code>DBGWVR<n>[31:3]:100</code> selected.
<code>BAS[5] == 1</code>	Byte at address <code>DBGWVR<n>[31:3]:101</code> selected.
<code>BAS[6] == 1</code>	Byte at address <code>DBGWVR<n>[31:3]:110</code> selected.

BAS value	Description
BAS[7] == 1	Byte at address DBGWVR<n> [31:3]:111 selected.

Table D2-14 Supported BAS values when the DBGWVRn address alignment is word

BAS value ^a	Description
BAS[0] == 1	Byte at address DBGWVR<n> [31:2]:00 selected.
BAS[1] == 1	Byte at address DBGWVR<n> [31:2]:01 selected.
BAS[2] == 1	Byte at address DBGWVR<n> [31:2]:10 selected.
BAS[3] == 1	Byte at address DBGWVR<n> [31:2]:11 selected.

a. [DBGWCR<n>](#).BAS[7:4] are RES0.

If the BAS field selects more than one byte, the bytes it selects must be contiguous. For watchpoint behavior when its BAS field selects non-contiguous bytes, see [Other usage constraints](#).

When programming the BAS field with anything other than 0b11111111, a debugger must also program [DBGWCR<n>](#).MASK to be 0b000000. See [Programming dependencies of the BAS and MASK fields](#).

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the [DBGWVR<n>](#)[31:3]. See [Example D2-2](#).

The following are some example configurations of the BAS field:

- To program a watchpoint to generate a Watchpoint exception on the byte at address 0x1003, program:
 - [DBGWVR<n>](#) with 0x1000.
 - [DBGWCR<n>](#)._EL1.BAS to be 0b00001000.
- To program a watchpoint to generate a Watchpoint exception on the bytes at addresses 0x2003, 0x2004 and 0x2005, program:
 - [DBGWVR<n>](#) with 0x2000.
 - [DBGWCR<n>](#)._EL1.BAS to be 0b00111000.
- If the address programmed into the [DBGWVR<n>](#) is doubleword-aligned:
 - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program [DBGWCR<n>](#).BAS to be 0b00001111.
 - To generate a Watchpoint exception when any byte in the word starting at address [DBGWVR<n>](#)[31:3]:100 is accessed, program [DBGWCR<n>](#).BAS to be 0b11110000.

Note

Arm deprecates programming a [DBGWVR<n>](#) with an address that is not doubleword-aligned.

D2.9.4.3 Programming a watchpoint with eight or more bytes

A debugger can use the *MASK* field, [DBGWCR<n>](#).MASK, to program a single watchpoint with a data address range. The data address range must meet all of the following criteria:

- It is a size that is both:
 - A power-of-two.
 - A minimum of eight bytes.

- A maximum of 2GB.
- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked.

If n least significant address bits are masked, the watchpoint generates a Watchpoint exception on all of the following:

- Address `DBGWVR<n>[31:n]:000...`
- Address `DBGWVR<n>[31:n]:111...`
- Any address between these two addresses.

For example, if the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between `DBGWVR<n>[31:4]:0000` and `DBGWVR<n>[31:4]:1111`, including these addresses.

Note

- The most significant bit cannot be masked. This means that the full address cannot be masked.
 - For watchpoint behavior when its MASK field is programmed with a reserved value, see [Reserved DBGWCR<n>.MASK values](#).
-

When masking address bits, a debugger must both:

- Program `DBGWCR<n>.BAS` to be `0b11111111`. See [Programming dependencies of the BAS and MASK fields](#).
- In the `DBGWVR<n>`, set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see [Other usage constraints](#).

D2.9.5 Watchpoint behavior for certain instruction classes

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Prefetch instructions.
- All data cache maintenance instructions except `DCIMVAC`.

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable watchpoints on an IMPLEMENTATION DEFINED subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache maintenance, address translation, and TLB maintenance instructions are IMPLEMENTATION DEFINED.

The access size of the IMPLEMENTATION DEFINED instruction cache maintenance, address translation, and TLB maintenance instructions that generate Watchpoint exceptions are IMPLEMENTATION DEFINED.

See also:

- [Watchpoint behavior on accesses by Store-Exclusive instructions](#).
- [Watchpoint behavior on accesses by DCIMVAC instructions](#).

D2.9.5.1 Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an Exclusives monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

D2.9.5.2 Watchpoint behavior on accesses by DCIMVAC instructions

It is IMPLEMENTATION DEFINED whether DCIMVAC operations can generate Watchpoint exceptions. If they can, they are treated as data stores. This means that for a watchpoint to match on an access caused by a DCIMVAC instruction, the debugger must program DBGWCR<n>.LSC to be one of the following:

- 10 Match on data stores only.
- 11 Match on data stores and data loads.

Note

For the size of data accesses performed by DCIMVAC instructions, see [Watchpoint data address comparisons](#). The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

D2.9.6 Watchpoint usage constraints

See the following:

- [Reserved DBGWCR<n>.{SSC, HMC, PAC} values.](#)
- [Reserved DBGWCR<n>.LBN values.](#)
- [Programming dependencies of the BAS and MASK fields.](#)
- [Reserved DBGWCR<n>.BAS values.](#)
- [Reserved DBGWCR<n>.MASK values.](#)
- [Other usage constraints.](#)

D2.9.6.1 Reserved DBGWCR<n>.{SSC, HMC, PAC} values

[Table D2-15](#) shows when particular combinations of DBGWCR<n>.{SSC, HMC, PAC} are reserved.

Table D2-15 Reserved SSC, HMC, and PAC combinations

HMC, SSC, and PAC combination	Reserved
Combinations not included in Table D2-12 .	Always

If a watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
 - Disabled.
 - Programmed with a combination that is not reserved, other than for a direct or external read of [DBGWCR<n>](#).
- For a direct or external read of [DBGWCR<n>](#), if the reserved combination:
 - Has no function for any execution conditions, the value read back for each of SSC, HMC, and PAC is UNKNOWN.
 - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of SSC, HMC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

D2.9.6.2 Reserved DBGWCR<n>.LBN values

For Linked watchpoints

A Linked watchpoint must link to a context-aware breakpoint. For a Linked watchpoint, any DBGWCR<n>.LBN value that is not for a context-aware breakpoint is reserved.

If a Linked watchpoint links to a breakpoint that is not implemented, or that is not context-aware, then reads of DBGWCR<n>.LBN return an UNKNOWN value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked watchpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN context-aware breakpoint.

If a Linked watchpoint links to a breakpoint that is implemented and is context-aware, but that is either not enabled or not programmed as a Linked Context breakpoint, it behaves as if it is disabled.

For Unlinked watchpoints

For Unlinked watchpoints, DBGWCR<n>.LBN reads UNKNOWN and its value is ignored.

D2.9.6.3 Programming dependencies of the BAS and MASK fields

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the DBGWVR<n> the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be 0b11111111, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be 0b000000, so that the MASK field does not indicate any address ranges.

If an enabled watchpoint has a MASK field that is nonzero and a BAS field that is not set to 0b11111111, then for each byte in the address range, it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint exception is generated.

D2.9.6.4 Reserved DBGWCR<n>.BAS values

The BAS field must be programmed with a value Zeros (8-n-m) : Ones (n) : Zeros (m), where:

- n is a nonzero positive integer less-than-or-equal-to 8.
- m is a positive integer less-than 8. m is permitted to be zero.
- n+m is less-than-or-equal-to 8.
- If DBGWVR<n>[2] is 1, then n+m is less-than-or-equal-to 4.

All other values are reserved.

Note

Zeros{0}() is an empty bitstring.

If DBGWVR<n>[2] is 1, DBGWCR<n>.BAS[7:4] are RES0.

If a watchpoint is programmed with a reserved BAS value:

- It is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the DBGWVR<n>.
- A direct or external read of DBGWCR<n>.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

D2.9.6.5 Reserved DBGWCR<n>.MASK values

If a watchpoint is programmed with a reserved MASK value:

- The watchpoint must behave as if it is either:
 - Disabled.
 - Programmed with an UNKNOWN value that is not reserved, that might be 0b00000, other than for a direct or external read of DBGWCR<n>.
- A direct or external read of DBGWCR<n>.MASK returns an UNKNOWN value.

D2.9.6.6 Other usage constraints

For all watchpoints:

- DBGWVR<n>[1:0] are RES0 and are ignored.
- If DBGWCR<n>.MASK is nonzero, and any masked bits of DBGWVR<n> are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if DBGWCR<n>.LSC is 0b00.

D2.9.7 Exception syndrome information, fault address information, and preferred return address

See the following:

- [Exception syndrome information](#).
- [Fault address information](#).
- [Preferred return address](#).

D2.9.7.1 Exception syndrome information

The PE takes a Watchpoint exception as either:

- A Data Abort exception, if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because HCR.TGE or HDCR.TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

Abort mode

The PE sets all of the following:

- DBGDSCRExt.MOE to 0b1010, to indicate a Watchpoint exception.
- DFSR.CM to indicate whether a cache maintenance instruction caused the exception.
- DFSR.WnR to indicate whether the exception was generated on a read instruction or a write instruction.

Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, HSR. See [Table D2-16](#).
- Sets DBGDSCRExt.MOE to 0b1001, to indicate a Watchpoint exception.

Table D2-16 Information recorded in the HSR

HSR field	Information recorded	
<i>Exception Class</i> , EC	The PE sets this to the code for a Data Abort exception routed to Hyp mode, 0×24 .	
<i>Instruction Length</i> , IL	The PE sets this to 1.	
<i>Instruction Specific Syndrome</i> , ISS	ISV[24]	<i>Instruction Syndrome Valid</i> (ISV). The PE sets this to 0.
	ISS[23:10]	RES0.
	ISS[9]	<i>External Abort type</i> (EA). The PE sets this to 0.
	ISS[8]	<i>Cache Maintenance</i> (CM). The PE sets this to indicate whether a cache maintenance instruction caused the exception.
	ISS[7]	RES0 .
	ISS[6]	<i>Write not Read</i> (WnR). The PE sets this to indicate whether the exception was generated on a read instruction or a write instruction .
	ISS[5:0]	<i>Data Fault Status Code</i> (DFSC). The PE sets this to the code for a debug exception, $0b100010$.

Note

For information about how debug exceptions can be routed to PL2, see [Routing debug exceptions](#).

D2.9.7.2 Fault address information

On a Watchpoint exception, the PE records an address in a *Fault Address Register* that the debugger can use to determine the memory location that triggered the watchpoint.

The Fault Address Register (FAR) used is one of the following:

- For a Watchpoint taken as an exception to Abort mode, [DFAR](#).
- For a Watchpoint taken as an exception to Hyp mode, [EHDFAR](#).
- For a Watchpoint debug event that causes entry to Debug state, [EDWAR](#).

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

Note

On entry into Debug state [EDWAR](#)[63:32] is UNKNOWN and must be ignored by the debugger.

In the following subsection, a *watchpointed address* is defined as any address for which at least one of the watchpoints generates a watchpoint match.

D2.9.7.2.1 Address recorded for Watchpoint exceptions generated by data cache maintenance instructions

For a watchpoint triggered by a Data Cache maintenance operation, the address recorded in [DFAR](#), [HDFAR](#), or [EDWAR](#) is both:

- Any address accessed by the instruction.
- Within a naturally-aligned block of memory.

D2.9.7.2.2 Address recorded for Watchpoint exceptions generated by instructions other than data cache maintenance instructions

The address recorded in [DFAR](#), [HDFAR](#), or [EDWAR](#) is both:

- From the inclusive range between:
 - The lowest address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
 - The highest watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.

- Within a naturally-aligned block of memory that is all of the following:
 - A power-of-two size.
 - No larger than the DC ZVA block size.
 - Contains a watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.

The size of the naturally-aligned block of memory is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

Example D2-3 Address recorded for a watchpoint programmed on 0x8019

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte 0x8019.

An A32 load multiple instruction then loads nine registers starting from address 0x8004 upwards. This triggers the watchpoint.

If the DC ZVA block size is:

- 32 bytes, the address that the PE records must be between 0x8004 and 0x8019 inclusive.
- 16 bytes, the address that the PE records must be between 0x8010 and 0x8019 inclusive.

D2.9.7.3 Preferred return address

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.9.8 Pseudocode description of Watchpoint exceptions taken from AArch32 state

`AArch32_WatchpointByteMatch()` tests an individual byte accessed by an operation.

`AArch32_StateMatch()` tests the values in `DBGWCR<n>`. {HMC, SSC, PAC}, and if the watchpoint is Linked, also tests the Linked Context breakpoint that the watchpoint links to.

`AArch32_WatchpointMatch()` tests the value in `DBGWVR<n>`.

`AArch32_CheckWatchpoint()` generates a `FaultRecord`. A Watchpoint exception is taken if all of the following are true:

- `DBGDSCRext.MDBGen` is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See [Enabling debug exceptions](#).
- All of the conditions required for Watchpoint exception generation are met. See [About Watchpoint exceptions](#).

Note

`AArch32_CheckWatchpoint` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

The `AArch32_Abort()` function processes the `FaultRecord` object returned by `AArch32_CheckWatchpoint()`, as described in [Abort exceptions](#). If a Watchpoint exception is taken to AArch32 state, the `AArch32_Abort()` function generates a Data Abort exception.

D2.10 Vector Catch exceptions

Arm deprecates the use of vector catch.

This section describes Vector Catch exceptions in stage 1 of an AArch32 translation regime.

This section contains the following subsections:

- [About Vector Catch exceptions.](#)
- [Exception vectors that Vector Catch exceptions can be enabled for.](#)
- [Generation of Vector Catch exceptions.](#)
- [Usage constraints.](#)
- [Exception syndrome information and preferred return address for a Vector Catch exception.](#)
- [Pseudocode description of Vector Catch exceptions.](#)

D2.10.1 About Vector Catch exceptions

Whenever the PE takes an exception, execution is forced to an address that is the *exception vector* for that exception. Vector catch permits a debugger to trap exceptions based on the exception vector, or based on the exception type associated with the exception vector, as follows:

- If the *address-matching* form of vector catch is implemented, the debugger can trap exceptions based on the exception vector.
- If the *exception-trapping* form of vector catch is implemented, the debugger can trap exceptions based on the exception type associated with the exception vector.

The architecture supports only these two forms of vector catch. Only one form can be implemented, and which is implemented is IMPLEMENTATION DEFINED. The [DBGDEVID](#) indicates which form is implemented.

Regardless of the form of vector catch implemented, a debugger enables Vector Catch exceptions for exception vectors or types by programming the [DBGVCR](#). This register contains *vector catch enable bits*. Each of these bits corresponds to a different vector. When a debugger sets a vector catch enable bit to 1, Vector Catch exceptions are enabled for the corresponding exception vector or type.

When Vector Catch exceptions are enabled for an exception vector, this is called an *enabled vector catch*. The set of exception vectors that Vector Catch exceptions are enabled for is called the *enabled vector catch set*.

If the form of vector catch implemented is the:

Address-matching form:

The PE compares the virtual address of each instruction in the program flow with a subset of the enabled vector catch set.

If an address match occurs, a Vector Catch exception is generated when the instruction that caused the match is committed for execution.

Exception-trapping form

Whenever the PE takes an exception, if the vector the exception is taken to is included in a subset of the enabled vector catch set, a Vector Catch exception is generated.

The Vector Catch exception is generated as part of entry to the exception, and must be taken before the PE either executes any instructions or takes any further exceptions.

[Table D2-17](#) summarizes the differences between the address-matching and exception-trapping forms.

Table D2-17 Differences in behavior of the address-matching and exception-trapping forms of vector catch

Address-matching	Exception-trapping
An enabled vector catch generates a Vector Catch exception when an instruction that is fetched from the vector is committed for execution. This means that spurious Vector Catch exceptions might occur, where the Vector Catch exception does not result from an exception entry, but is instead caused by a branch to the vector. A branch to the vector might occur, for example, on a return from a nested exception or when simulating an exception entry.	An enabled vector catch generates a Vector Catch exception immediately after the PE takes the exception that is associated with the vector. This means that Vector Catch exceptions always result from exception entry, and not from branches to exception vectors.
A Vector Catch exception is generated as a result of an instruction fetch. This means that the Vector Catch exception has a priority relative to the other synchronous exceptions that result from an instruction fetch.	A Vector Catch exception is generated as a result of an exception entry. This means that the Vector Catch exception is part of the exception that caused the Vector Catch exception. Therefore, the Vector Catch exception has no priority associated with it.
A Vector Catch exception can be preempted by another exception. If this happens, the Vector Catch exception is generated again when the exception handler branches back to the vector.	Vector Catch exceptions must be taken before other exceptions.
A Vector Catch exception can be generated as a result of an instruction fetch executed in any AArch32 mode except Hyp mode, including User mode.	Because a Vector Catch exception is generated as the result of an exception entry, the Vector Catch exception is taken.
If HCR.TGE is 1, Vector Catch exceptions can be generated for User mode instruction fetches from PL1 vectors.	If HCR.TGE is 1, Vector Catch exceptions are never generated.

Depending on the implementation, some vector catch enable bits in the [DBGVCR](#) might be RES0.

See [Exception vectors that Vector Catch exceptions can be enabled for](#) for the vector catch enable bits that exist for different implementations.

[The debug exception enable controls](#) describes the enable controls for Vector Catch exceptions.

D2.10.2 Exception vectors that Vector Catch exceptions can be enabled for

When the PE takes an exception, the exception vector is contained in a *vector table* at the Privilege level the exception is taken to.

Depending on the Security state and AArch32 mode the exception is taken to, when the exception is taken, the vector table used is the table that contains one of:

- *Local exception vectors.*
- *Hyp exception vectors.*

The tables that follow show the vectors that Vector Catch exceptions can be enabled for, and their corresponding vector catch enable bits in the [DBGVCR](#):

- [Table D2-18](#) shows the Local exception vectors that Vector Catch exceptions can be enabled for.

The architecture does not provide vector catch enable bits for the Hyp exception vectors.

Table D2-18 Local exception vectors that Vector Catch exceptions can be enabled for

Vector catch enable bit	Exception type	Local exception vectors
Local exception vectors		
F	FIQ interrupt	$\text{VBAR} + 0 \times 0000001C$
I	IRQ interrupt	$\text{VBAR} + 0 \times 00000018$
D	Data Abort	$\text{VBAR} + 0 \times 00000010$
P	Prefetch Abort	$\text{VBAR} + 0 \times 0000000C$
S	Supervisor Call	$\text{VBAR} + 0 \times 00000008$
U	Undefined Instruction	$\text{VBAR} + 0 \times 00000004$

D2.10.3 Generation of Vector Catch exceptions

How Vector Catch exceptions are generated depends on which form is implemented:

- [Address-matching form.](#)
- [Exception-trapping form.](#)

D2.10.3.1 Address-matching form

The PE compares the virtual address of each instruction in the program flow is with some or all of the addresses in the enabled vector catch set, as follows:

The enabled vector catch set contains only Local exception vectors.

The PE compares the virtual address of each instruction in the program flow, including those executed at EL0, with all addresses in the enabled vector catch set.

For each instruction in the program flow, the PE tests for any possible Vector Catch exceptions before executing the instruction. If a match occurs, a Vector Catch exception is generated when the instruction is committed for execution, regardless of all of the following:

- Whether the instruction passes its Condition code check.
- Whether the instruction is executed as part of exception entry.
- What [HCR](#).{IMO, FMO, AMO} are set to.

D2.10.3.2 Exception-trapping form

When the PE takes an exception, it tests whether the exception is by branching to an exception vector in a subset of the enabled vector catch set, as follows:

The enabled vector catch set contains only Local exception vectors.

The PE tests whether the exception is by branching to any address in the enabled vector catch set.

If the exception is by branching to a vector in the subset, a Vector Catch exception is generated as part of exception entry. That is, a Vector Catch exception is generated instead of the exception handler executing its first instruction.

D2.10.4 Usage constraints

See the following subsections:

- [Usage constraints that apply to both forms of vector catch.](#)
- [Usage constraints that apply only to the address-matching form.](#)

D2.10.4.1 Usage constraints that apply to both forms of vector catch

For Vector Catch exceptions enabled for either the Prefetch Abort exception vector or the Data Abort exception vector, if one of these exception types is taken to the Exception level that debug exceptions are targeting, behavior is **CONSTRAINED UNPREDICTABLE**. Either:

- Vector catch is ignored, therefore a Vector Catch exception is not generated.
- Vector catch generates a Prefetch Abort debug exception. For Vector Catch exceptions enabled for the Prefetch Abort exception vector, the PE might enter a recursive loop of Prefetch Abort exceptions causing Vector Catch exceptions and Vector Catch exceptions causing Prefetch Abort exceptions.

Note

The Exception level that debug exceptions are targeting is called the *debug target Exception level*, EL_D . [Routing debug exceptions](#) describes how EL_D is derived.

D2.10.4.2 Usage constraints that apply only to the address-matching form

Exception vectors are at word-aligned addresses, and:

- It is **CONSTRAINED UNPREDICTABLE** whether an enabled vector catch generates a Vector Catch exception for a 32-bit T32 instruction starting at the halfword-aligned address immediately prior to the vector address.
- T32 instructions that start at the halfword-aligned address immediately after the exception vector do not generate Vector Catch exceptions.

For the address-matching form, Vector Catch exceptions have the same priority as Breakpoint exceptions. If a single instruction causes both a Vector Catch exception and a Breakpoint exception, it is **CONSTRAINED UNPREDICTABLE** which of these debug exceptions the PE takes.

D2.10.5 Exception syndrome information and preferred return address for a Vector Catch exception

See the following:

- [Exception syndrome information for a Vector Catch exception](#).
- [Preferred return address for a Vector Catch exception](#).

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode. The referenced section uses the term more generally, to include exception information reported in the [IFSR](#).

D2.10.5.1 Exception syndrome information for a Vector Catch exception

The PE takes a Vector Catch exception as either:

- A Prefetch Abort exception if it is taken to PL1. In this case, it is taken to Abort mode.
- A Hyp trap exception, if it is taken to PL2 because [HCR](#).TGE or [HDCR](#).TDE is 1. In this case, it is taken to Hyp mode.

If the exception is taken to:

PL1 Abort mode

The PE sets all of the following:

- [IFSR](#).FS to the code for a debug exception, 0b00010.
- [DBGDSCRExt](#).MOE to 0b0101, to indicate a Vector Catch exception.
- The [IFAR](#) with an UNKNOWN value.

PL2 Hyp mode

The PE does all of the following:

- Records information about the exception in the *Hypervisor Syndrome Register*, [HSR](#). See [Table D2-19](#).
- Sets [DBGDSCRExt.MOE](#) to 0b0101, to indicate a Vector Catch exception.
- Sets the [HIFAR](#) to an unknown value.

Table D2-19 Information recorded in the HSR

HSR field	Information recorded	
Exception Class, EC	The PE sets this to the code for a Prefetch Abort exception routed to Hyp mode, 0x20.	
Instruction Length, IL	The PE sets this to 1.	
Instruction Specific Syndrome, ISS	ISS[24:10]	RES0.
	ISS[9]	External Abort type (EA). The PE sets this to 0.
	ISS[8:6]	RES0.
	ISS[5:0]	Instruction Fault Status Code (IFSC). The PE sets this to the code for a debug exception, 0b100010.

Note
For information about how debug exceptions can be routed to PL2, see [Routing debug exceptions](#).

D2.10.5.2 Preferred return address for a Vector Catch exception

The preferred return address of a Vector Catch exceptions is the address of the instruction that was not executed because the PE took the Vector Catch exception instead.

This means that the preferred return address is the exception vector. This is true regardless of whether the address-matching form or the exception trapping form is implemented.

D2.10.6 Pseudocode description of Vector Catch exceptions

The `AArch32_VCRMatch()` pseudocode function checks whether the instruction at `address` generates a Vector Catch exception. It therefore shows the address-matching form of vector catch.

The `AArch32_CheckVectorCatch()` pseudocode function uses `AArch32_VCRMatch()` to test whether the instruction generates a Vector Catch exception, and if `AArch32_VCRMatch()` returns TRUE it generates that event.

The `AArch32_Abort()` function processes the `FaultRecord` object returned by `AArch32_CheckVectorCatch()`, as described in [Abort exceptions](#). If there is a Vector Catch exception, the `AArch32_Abort()` function generates a Prefetch Abort exception.

D2.11 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
 - External debug registers.
 - System registers, including system debug registers.
 - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a [Context Synchronization event](#) has occurred.

For any instructions executed between the time when the change is made and the time when the next [Context Synchronization event](#) occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

Example D2-4 Example of synchronization and Breakpoint exception generation

1. Software changes [DBGDSCRext.MDBGGen](#) from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A [Context Synchronization event](#) occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

Example D2-5 Example of synchronization and debug exceptions generation

1. Software unlocks the OS Lock.
2. The PE executes some instructions.
3. A [Context Synchronization event](#) occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Breakpoint Instruction exceptions can be generated.

Note

For more information, see:

- [Synchronization of changes to AArch32 System registers.](#)
 - [Accessing PSTATE fields.](#)
 - [Synchronization of changes to the external debug registers.](#)
-

D2.11.1 Mode changes without explicit Context Synchronization events

Most changes to the Exception level, happen as a result of operations that are an explicit [Context Synchronization event](#). This is because taking an exception and returning from an exception are both explicit [Context Synchronization events](#), and the Privilege level can only change as a result of taking or returning from an exception.

However, some AArch32 mode changes can happen because of operations that are not an explicit [Context Synchronization event](#).

These are AArch32 mode changes caused by MSR and CPS instructions.

A mode change might be to a mode at a lower Privilege level.

Chapter D3

AArch32 Trace

This chapter describes the AArch32 self-hosted trace:

- [About trace](#)
- [Prohibited regions in self-hosted trace](#)
- [Self-hosted trace timestamps](#)
- [Synchronization in self-hosted trace](#)

D3.1 About trace

A trace unit generates trace data to describe the program flow of the PE.

The trace unit can be an implementation of any of the following:

- In Armv8, a standard Arm Embedded Trace Macrocell (ETM). See *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4*.
- An IMPLEMENTATION DEFINED trace function.

Self-hosted trace happens when the agent controlling the trace collection is part of the same software stack as the software being traced. The agent controls prohibited regions. The information collected by the agent is sent to a trace sink.

The trace unit and the PE must have the same view of the debug authentication interface.

`ExternalNoninvasiveDebugAllowed()` controls whether tracing is prohibited or allowed.

D3.1.1 Trace Sinks

The trace unit sends the trace data to a trace sink. A system might include multiple trace sinks, and allow software to configure which trace sink or sinks are used.

An example of an internal trace sink is an Embedded Trace Router (ETR), which allows software to define a buffer in memory. Trace data is written to this buffer.

D3.2 Prohibited regions in self-hosted trace

Trace is not generated in prohibited regions. The pseudocode function `TraceAllowed()` indicates whether tracing is allowed in the current Security state and Exception level.

The IMPLEMENTATION DEFINED debug authentication interface can allow an external agent to disable the self-hosted trace extension.

D3.3 Self-hosted trace timestamps

The trace timestamp is a value that represents the passage of time in real-time. It is calculated from a counter which increments all the time, when the PE is generating trace and when the PE is in a prohibited region.

While self-hosted trace is disabled, the external trace provides the trace timestamp. If the external trace is a standard CoreSight system, the relationship between CoreSight time and the Generic Timer counter is IMPLEMENTATION DEFINED.

When self-hosted trace is enabled, the trace time stamp is one of the following:

- Physical time, which is defined by the physical count value returned by `PhysicalCountInt()`.
- Virtual time, which is defined as the value of `PhysicalCountInt()` minus a virtual offset.
The virtual offset is always `CNTVOFF`, including when a read of `CNTVCT` at the current Exception level would treat the virtual offset as zero.

D3.4 Synchronization in self-hosted trace

The trace unit is an indirect observer of the trace control registers.

Chapter D4

The AArch32 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:

- [About the memory system architecture](#)
- [Address space](#)
- [Mixed-endian support in AArch32](#)
- [AArch32 cache and branch predictor support](#)
- [System register support for IMPLEMENTATION DEFINED memory features](#)
- [External aborts](#)
- [Memory barrier instructions](#)
- [Pseudocode description of general memory System instructions](#)

D4.1 About the memory system architecture

The Arm architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system, see [Memory hierarchy](#).

D4.1.1 Form of the memory system architecture

The Armv8 R-profile AArch32 architecture includes a *Protected Memory System Architecture* (PMSAv8-32). [The Protected Memory System Architecture](#) describes the PMSAv8-32.

D4.1.2 Memory attributes

[Memory types and attributes](#) describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

[Table D4-1](#) shows the memory attributes that are visible at the system level.

Table D4-1 Memory attribute summary

Memory type	Shareability	Cacheability
Device ^a	Outer Shareable.	Non-cacheable.
Normal	One of: - Non-shareable. - Inner Shareable. - Outer Shareable.	One of ^b : - Non-cacheable. - Write-Through Cacheable. - Write-Back Cacheable.

a. Takes additional attributes, see [Device memory](#).
b. See also [Cacheability](#), [cache allocation hints](#), and [cache transient hints](#).

For more information on Cacheability and Shareability, see [The Cacheability and Shareability memory attributes](#), [Non-shareable Normal memory](#), and [Caches and memory hierarchy](#).

D4.2 Address space

The architecture is designed to support a wide range of applications with different memory requirements. It supports a range of *Physical Address* (PA) sizes, and provides associated control and identification mechanisms. For more information, see [About PMSAv8-32](#).

D4.2.1 Address space overflow or underflow

This subsection describes address space overflow or underflow:

D4.2.1.1 Instruction address space overflow

When a PE performs a normal, sequential execution of instructions, it calculates:

$$\text{address_of_next_instruction} = \text{address_of_current_instruction} + \text{size_of_executed_instruction}$$

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an A32 or T32 instruction overflows `0xFFFF_FFFF`, the Program Counter becomes UNKNOWN.

If the PE executes an instruction for which the instruction address, size, and alignment mean that it contains the bytes `0xFFFF_FFFF` and `0x0000_0000`, the bytes that apparently from `0x0000_0000` onwards come from an UNKNOWN address.

D4.2.1.2 Data address space overflow and underflow

If the PE executes a load or store instruction for which the calculated address, total access size, and alignment mean that it accesses bytes `0xFFFF_FFFF` and `0x0000_0000`, then the bytes that apparently come from `0x0000_0000` onwards come from UNKNOWN addresses.

D4.3 Mixed-endian support in AArch32

Table D4-2 shows the endianness of explicit data accesses and translations.

Table D4-2 Endianness support

Exception level	Explicit data accesses	Stage 1 translations	Stage 2 translations
EL0	PSTATE.E	SCTLR.EE	HSCTLR.EE
EL1	PSTATE.E	SCTLR.EE	HSCTLR.EE
EL2	PSTATE.E	HSCTLR.EE	n/a

The architecture provides the following options for endianness support:

- All Exception levels support mixed-endianness:
 - [SCTLR.EE](#), [HSCTLR.EE](#), and [PSTATE.E](#) are RW.
- Only EL0 supports mixed-endianness and EL1 and EL2 support only little-endianness:
 - [SCTLR.EE](#) and [HSCTLR.EE](#) are RES0. [PSTATE.E](#) is RW when in EL0 and RES0 when in EL1 or EL2. [SPSR.E](#) is also RES0 when not returning to EL0.
- Only EL0 supports mixed-endianness and EL1 and EL2 support only big-endianness:
 - [SCTLR.EE](#) and [HSCTLR.EE](#) are RES1. [PSTATE.E](#) is RW when in EL0 and RES1 when in EL1 or EL2. [SPSR.E](#) is also RES1 when not returning to EL0.
- All Exception levels support only little-endianness:
 - Each of [SCTLR.EE](#), [HSCTLR.EE](#), [PSTATE.E](#), and [SPSR.E](#) is RES0.
- All Exception levels support only big-endianness:
 - Each of [SCTLR.EE](#), [HSCTLR.EE](#), [PSTATE.E](#), and [SPSR.E](#) is RES1.

If mixed endian support is implemented for an Exception level, endianness is controlled by [PSTATE.E](#). For exception returns, [PSTATE.E](#) is copied from [SPSR.E](#). If the target Exception level supports only little-endian accesses, [SPSR.E](#) is RES0. If the target Exception level supports only big-endian accesses, [SPSR.E](#) is RES1.

———— Note ————

Arm deprecates [PSTATE.E](#) having a different value from the equivalent System register EE bit when in EL1 or EL2. The use of the `SETEND` instruction is also deprecated.

The `BigEndian()` function determines whether the current Exception level and Execution state is using big-endian data.

For more information about endianness in the Arm architecture, see [Endian support](#).

D4.4 AArch32 cache and branch predictor support

The following sections describe the support for caches and branch predictors in AArch32 state:

- [General behavior of the caches.](#)
- [Cache identification.](#)
- [Cacheability, cache allocation hints, and cache transient hints.](#)
- [Enabling and disabling the caching of memory accesses in AArch32 state.](#)
- [Behavior of caches at reset.](#)
- [About cache maintenance in AArch32 state.](#)
- [AArch32 cache and branch predictor maintenance instructions.](#)
- [Cache lockdown.](#)
- [System level caches.](#)

See also [The Protected Memory System Architecture](#), and in particular [PMSAv8-32 implications for caches](#).

Note

- Branch predictors typically use a form of cache to hold branch target data. Therefore, they are included in this section.
 - In the instruction mnemonics, MVA is a synonym for VA.
-

D4.4.1 General behavior of the caches

When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
 - A particular implementation.
 - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

Note

For more information, see [The interaction of cache lockdown with cache maintenance instructions](#).

- Any memory location that has a Normal Cacheable attribute at either the current Exception level or at a higher Exception level can be allocated to a cache at any time.
- It is guaranteed that no memory location that does not have a Normal Cacheable attribute is allocated into the cache.
- It is guaranteed that no memory location is allocated to the cache if it has a Normal Non-cacheable attribute or any type of Device memory attribute in both:
 - The translation regime at the current Exception level.
 - The translation regime at any higher Exception level.
- For data accesses, any memory location with a Normal Inner Shareable or Normal Outer Shareable attribute is guaranteed to be coherent with all Requesters in its Shareability domain.
- Any memory location is not guaranteed to remain incoherent with the rest of memory.
- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the Shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the *Cache Write-back Granule*. In some implementations the [CTR](#) identifies the Cache Write-back Granule.
- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it was previously visible to that observer.

Note

The Cacheability attribute of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the [SCTLR](#).{I, C} controls.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to the size of the cache entry.

D4.4.2 Cache identification

The cache identification consists of a set of registers that describe the implemented caches that are affected by cache maintenance instructions executed on the PE. This includes cache maintenance instructions that:

- Affect the entire cache, for example [ICIALLUIS](#).
- Operate by VA, for example [ICIMVAU](#).
- Operate by set/way, for example [DCISW](#).

The cache identification registers are:

- A single Cache Type Register, [CTR](#), that defines:
 - The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
 - The minimum line length of any of the data or unified caches, affected by the data cache maintenance instructions.
 - The cache indexing and tagging policy of the Level 1 instruction cache.

Note

It is IMPLEMENTATION DEFINED whether caches beyond the PoC will be reported by this mechanism, and because of the possible existence of system caches some caches before the PoC might not be reported. For more information about system caches, see [System level caches](#).

- A single Cache Level ID Register, [CLIDR](#), that defines:
 - The type of cache that is implemented and can be maintained using the architected cache maintenance instructions that operate by set/way or operate on the entire cache at each cache level, up to the maximum of seven levels.

- The *Level of Unification Inner Shareable* (LoUIS), *Level of Coherence* (LoC) and the *Level of Unification* (LoU) for the caches. See [Terms used in describing the cache maintenance instructions](#) for a definition of these terms.
- An optional ICB field to indicate the boundary between the caches use for caching Inner Cacheable memory regions and those used only for caching Outer Cacheable regions.
- A single Cache Size Selection Register, [CSSELR](#), that selects the cache level and sort of cache (Instruction, Data/Unified/Tag) of the current Cache Size Identification Register.
- For each implemented cache that is identifiable by this mechanism, across all the levels of caching, a Cache Size Identification Register, that defines:
 - The number of sets, associativity, and line length of the cache. See [Terms used in describing the cache maintenance instructions](#) for a definition of these terms.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.
2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.
3. For each cache identified at stage 2:
 - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
 - An instruction cache.
 - A data or unified cache.
 - Read the Cache Size Identification Register to find details of the cache.
 - There is a single 32-bit Cache Size Identification Register, [CCSIDR](#).
 - The length of the [CCSIDR](#).Assoc field is 10 bits. This limits the associativity of the currently selected cache to 2^{10} .
 - The length of the [CCSIDR](#).NumSets field is 15 bits. This limits the number of sets in the currently selected cache to 2^{15} .

D4.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability applies only to Normal memory, and is defined independently for Inner and Outer cache locations. All types of Device memory are always treated as Non-cacheable.

As described in [Memory types and attributes](#), the memory attributes include a Cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

Cacheability attributes other than Non-cacheable can be complemented by a *cache allocation hint*. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. In addition, it is IMPLEMENTATION DEFINED whether a *cache transient hint* is supported, see [Transient Cacheability hint](#).

The cache allocation hints are assigned independently for read and write accesses, and therefore when the Transient hint is supported the following cache allocation hints can be used:

For read accesses: Read-Allocate, Transient Read-Allocate, or No Read-Allocate.

For write accesses: Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

Note

- A Cacheable location with both No Read-Allocate and No Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, No Read-Allocate, No Write-Allocate.
 - Implementations can use the cache allocation hints to limit cache pollution to a part of a cache, such as to a subset of ways.
 - For translations, the appropriate ATTRindx fields define the memory attributes of the translation, including the Cacheability.
-

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

D4.4.3.1 Transient Cacheability hint

It is IMPLEMENTATION DEFINED whether a Transient hint is supported for the translation scheme. In an implementation that supports the Transient hint, the Transient hint is a qualifier of the cache allocation hints, and indicates that the benefit of caching is for a relatively short period. It indicates that it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

Note

The architecture does not specify what is meant by a *relatively short period*.

The description of the [MAIR0](#), [MAIR1](#), [HMAIR0](#), and [HMAIR1](#) registers includes the assignment of the Transient attribute in an implementation that supports this option. In this assignment:

- The Transient hint is defined independently for Inner Cacheable and Outer Cacheable memory regions.
- A single Transient hint applies to both read and write accesses to a memory region.

D4.4.4 Enabling and disabling the caching of memory accesses in AArch32 state

Cacheability control fields can force all memory locations with the Normal memory type to be treated as Non-cacheable, regardless of their assigned Cacheability attribute. Independent controls are provided for each stage of address translation, with separate controls for:

- Data accesses. These controls also apply to accesses to the translation tables.
- Instruction accesses.

Note

These Cacheability controls replace the cache enable controls provided in previous versions of the Arm architecture.

In AArch32 state, the Cacheability control fields and their effects are as follows:

For the PL1&0 translation regime

[SCTLR](#) holds the EL1 controls that affect Cacheability:

- When the value of [SCTLR.C](#) is 0:
 - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
 - All accesses to the PL1&0 stage 1 translation tables are Non-cacheable.
- When the value of [SCTLR.I](#) is 0:
 - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of [HCR.DC](#) is 1, all stage 1 translations and all accesses to the EL1&0 stage 1 translation tables, are treated as accesses to Normal Non-shareable Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate memory, regardless of the value of [SCTLR.C](#). This applies to translations for both data and instruction accesses.

In addition, when the value of **SCTLR.M** is 0, indicating that the stage 1 translations are disabled for the translation regime, if the value of **HCR.DC** is 0 then:

- If the value of **SCTLR.I** is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of **SCTLR.I** is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

- The stage 1 and stage 2 Cacheability attributes are combined as described in Combining the Cacheability attribute.
 - **SCTLR.{C, I}** and **HCR.DC** fields have no effect on the EL2 translation regimes.
-

For the EL2 translation regime

- When the value of **HSCTLR.C** is 0:
 - All data accesses to Normal memory using the EL2 translation regime are Non-cacheable.
 - All accesses to the EL2 translation tables are Non-cacheable.
- When the value of **HSCTLR.I** is 0:
 - All instruction accesses to Normal memory using the EL2 translation regime are Non-cacheable.

In addition, when the value of **HSCTLR.M** is 0, indicating that stage 1 translations are disabled, then:

- If the value of **HSCTLR.I** is 0, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.
- If the value of **HSCTLR.I** is 1, instruction accesses to Normal memory from stage 1 of the translation regime are Outer Shareable, Inner Write-Through cacheable, Outer Write-Through cacheable.

Note

- The **HSCTLR.{I, C, M}** fields have no effect on the PL1&0 translation regime.
-

The effect of the **SCTLR.C** or **HSCTLR.C** bits is reflected in the result of the address translation instructions in the PAR.

Note

- The requirements in this section mean the architecturally required effects of **SCTLR.I** and **HSCTLR.I** are limited to their effects on caching instruction accesses in unified caches.
 - This specification can give rise to different Cacheability attributes between instruction and data accesses to the same location. Where this occurs, the measures for mismatch memory attributes described in [Mismatched memory attributes](#) must be followed to manage the corresponding loss of coherency.
-

D4.4.5 Behavior of caches at reset

The following rules apply to caches at reset:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- The Cacheability control fields described in [Enabling and disabling the caching of memory accesses in AArch32 state](#) reset to values that force all memory locations to be treated as Non-cacheable.

Note

This applies only to the controls that apply to the Translation regime that is used by the Exception level and PE mode entered on reset.

- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before caching is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.

- If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable then the cache initialization routine must:
 - Provide a mechanism to ensure the correct initialization of the caches.
 - Be documented clearly as part of the documentation of the device.

In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.

- Arm recommends that whenever an invalidation routine is required, it is based on the cache maintenance instructions.

Similar rules apply to:

- Branch predictor behavior, see [Behavior of the branch predictors at reset](#).

D4.4.6 About cache maintenance in AArch32 state

The following sections give general information about cache maintenance:

- [Terms used in describing the cache maintenance instructions](#).
- [Abstraction of the cache hierarchy](#).

The following sections describe the AArch32 state cache maintenance instructions:

- [AArch32 instruction cache maintenance instructions \(IC*\)](#).
- [AArch32 data cache maintenance instructions \(DC*\)](#).

Note

Some descriptions of the cache maintenance instructions refer to the Cacheability of the address on which the instruction operates. The Cacheability of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the [SCTLR](#).{I, C} controls.

D4.4.6.1 Terms used in describing the cache maintenance instructions

Cache maintenance instructions are defined to act on particular memory locations. Instructions can be defined:

- By the virtual address of the memory location to be maintained, referred to as operating *by VA*.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating *by set/way*.

In addition, for instruction caches and branch predictors, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- [Terminology for cache maintenance instructions operating by set/way](#).
- [Terminology for Clean, Invalidate, and Clean and Invalidate instructions](#).

Note

There is no terminology specific to cache maintenance instructions that operate by VA. When all applicable stages of translation are disabled, the VA used is identical to the PA. For more information about memory system behavior when address translation is disabled, see [PMSAv8-32 implications for caches](#).

D4.4.6.1.1 Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

Level	The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED. The cache levels that can be managed using the architected cache maintenance instructions that operate by set/way can be determined from the CLIDR. In the Arm architecture, the lower numbered cache levels are those closest to the PE. See Memory hierarchy.
Set	Each level of a cache is split up into a number of <i>sets</i>. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address. In the Arm architecture, sets are numbered from 0.
Way	The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The <i>way</i> number specifies one of these locations. In the Arm architecture, ways are numbered from 0.

Note

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, Arm expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.

D4.4.6.1.2 Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.
2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

The *Clean* and *Invalidate* instructions address these two issues. The definitions of these instructions are:

Clean	A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification. The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the Shareability domain of that memory location.
Invalidate	A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made. If the address of an entry on which the invalidate instruction operates is Normal, Non-cacheable or any type of Device memory then an invalidate instruction also ensures that this address is not present in the cache.

Note

Entries for addresses that are Normal Cacheable can be allocated to the cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in a cache.

Clean and Invalidate

A cache *clean and invalidate* instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.
- For instruction operating by VA, two conceptual points are defined:

Point of Coherency (PoC)

The point at which all observers in the Full System domain that can access a memory location are guaranteed to see the same copy of that memory location for accesses of any memory type or Cacheability attribute. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

Note

The presence of system caches can affect the determination of the Point of Coherency as described in [System level caches](#).

Point of Unification (PoU)

The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable Shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable Shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

1. Clean data cache entry by address.
2. Invalidate instruction cache entry by address.

The following fields in the [CLIDR](#) relate to these conceptual points:

LoC, Level of Coherence

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is 0x0, this means that no levels of cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

LoUU, Level of Unification, uniprocessor

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is 0x0, this means that no levels of cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification.

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

LoUIS, Level of Unification, Inner Shareable

In any implementation:

- This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain. As with LoC, the LoUIS value is a cache level.

- If the LoUIS field value is 0x0, this means that no levels of cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable Shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

For more information, see the [CLIDR](#) description.

D4.4.6.2 Abstraction of the cache hierarchy

The following subsections describe the abstraction of the cache hierarchy:

- [Cache maintenance instructions that operate by VA.](#)
- [Cache maintenance instructions that operate by set/way.](#)

D4.4.6.2.1 Cache maintenance instructions that operate by VA

The VA-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being either:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.

See [Terms used in describing the cache maintenance instructions](#) for definitions of Point of Coherency and Point of Unification, and more information about possible meanings of VA.

[AArch32 cache and branch predictor maintenance instructions](#) lists the VA-based maintenance instructions.

The [CTR](#) holds minimum line length values for:

- The instruction caches.
- The data and unified caches.

These values support efficient invalidation of a range of addresses, because this value is the most efficient address stride to use to apply a sequence of VA-based maintenance instructions to a range of VAs.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the [CTR](#) defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

D4.4.6.2.2 Cache maintenance instructions that operate by set/way

[AArch32 cache and branch predictor maintenance instructions](#) lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.
- An invalidate instruction invalidates only at the level specified.

D4.4.7 AArch32 cache and branch predictor maintenance instructions

The instruction and data cache maintenance instructions have the same functionality in AArch32 state and in AArch64 state. [Table D4-3](#) shows the AArch32 System instructions. Instructions that take an argument include Rt in the instruction description.

The architecture does not define any branch predictor maintenance instructions for AArch32 state.

Note

- In [Table D4-3](#) the Point of Unification is the Point of Unification of the PE executing the cache maintenance instruction.
- In AArch32 state, all of the maintenance instructions are available from EL1 or higher.

Table D4-3 AArch32 System instructions for cache maintenance

Register	Instruction
Instruction cache maintenance instructions	
ICIALLUIS	Invalidate all to Point of Unification, Inner Shareable
ICIALLU	Invalidate all to Point of Unification
ICIMVAU , Rt	Invalidate by virtual address to Point of Unification
Data cache maintenance instructions	
DCIMVAC , Rt	Invalidate by virtual address to Point of Coherency
DCISW , Rt	Invalidate by set/way
DCCMVAC , Rt	Clean by virtual address to Point of Coherency
DCCSW , Rt	Clean by set/way
DCCMVAU , Rt	Clean by virtual address to Point of Unification
DCCIMVAC , Rt	Clean and invalidate by virtual address to Point of Coherency
DCCISW , Rt	Clean and invalidate by set/way
Branch prediction maintenance instructions	
BPIMVA , Rt	Invalidate the virtual address from the branch predictors
BPIALLIS , Rt	Invalidate all entries from branch predictors, Inner Shareable
BPIALL , Rt	Invalidate all entries from branch predictors

A DSB instruction intended to ensure the completion of cache or branch predictor maintenance instructions must have an access type of both loads and stores.

In an implementation where the branch predictors are architecturally invisible, the [BPIMVA](#), [BPIALLIS](#), and [BPIALL](#) instructions can execute as NOPs.

The following subsections give more information about these instructions:

- [AArch32 instruction cache maintenance instructions \(IC*\)](#).
- [AArch32 data cache maintenance instructions \(DC*\)](#).
- [Branch predictors](#).
- [General requirements for the scope of cache and branch predictor maintenance instructions](#).
- [Effects of instructions that operate by VA to the Point of Coherency](#).
- [Effects of instructions that operate by VA but not to the Point of Coherency](#).
- [Effects of All and set/way maintenance instructions](#).
- [Effects of virtualization and security on the AArch32 cache maintenance instructions](#).
- [Boundary conditions for cache maintenance instructions](#).
- [Ordering of cache and branch predictor maintenance instructions](#).
- [Performing cache maintenance instructions](#).

D4.4.7.1 AArch32 instruction cache maintenance instructions (IC*)

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- Whether the instruction was performed from Hyp mode, or at EL1.
- For an instruction executed at EL1, the VMID.

That VA to PA translation might fault. However for an instruction cache maintenance instruction that operates by VA:

- It is IMPLEMENTATION DEFINED whether the operation can generate a Data Abort exception for a [Translation fault](#).
- The operation cannot generate a Data Abort exception for a [Permission fault](#).

For more information about the possible faults on an instruction that operates by VA, see [Types of MPU faults](#).

An instruction cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction. See also the completion requirements for cache and branch predictor maintenance instructions in [Completion and endpoint ordering](#).

See also [Ordering of cache and branch predictor maintenance instructions](#).

D4.4.7.2 AArch32 data cache maintenance instructions (DC*)

Data cache maintenance instructions that take a set/way/level argument take a 32-bit register.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see [Out of range values of the Set/Way/Index fields in cache maintenance instructions](#) or the instruction description.

DCISW instructions executed at EL1 perform a clean and invalidate, meaning it performs the same maintenance as a DCCISW instruction, if the value of HCR.SWIO is 1.

Where an address argument for these instructions is required, it takes the form of a 32-bit register that holds the virtual address argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- Whether the instruction was performed from Hyp mode, or from EL1.
- For an instruction executed from EL1, the VMID.

That VA to PA translation might fault.

For more information about the possible faults on an instruction that operates by VA, see [Types of MPU faults](#).

DCIMVAC and DCISW instructions executed at EL1 perform a clean and invalidate, meaning they perform the same maintenance as a DCCIMVAC or DCCISW instruction respectively, if both of the following apply:

- PL1&0 stage two address translation is enabled.
- The value of HCR.VM is 1.

If a memory fault that sets FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

See also [Ordering of cache and branch predictor maintenance instructions](#).

D4.4.7.3 Branch predictors

In AArch32 state it is IMPLEMENTATION DEFINED whether branch prediction is architecturally visible. This means that under some circumstances software must perform branch predictor maintenance to avoid incorrect execution caused by out-of-date entries in the branch predictor. For example, to ensure correct operation it might be necessary to invalidate

branch predictor entries on a change to instruction memory, or a change of instruction address mapping. For more information, see [Specific requirements for branch predictor maintenance instructions](#).

In an implementation where the branch predictors are architecturally invisible, the branch predictor maintenance instructions can execute as `NOPS`.

An invalidate all operation on the branch predictor ensures that any location held in the branch predictor has no functional effect on execution. An invalidate branch predictor by VA instruction operates on the address of the branch instruction, but can affect other branch predictor entries.

Note

The architecture does not make visible the range of addresses in a branch predictor to which the invalidate operation applies. This means the address used in the invalidate by VA operation must be the address of the branch to be invalidated.

If branch prediction is architecturally visible, an instruction cache invalidate all operation also invalidates all branch predictors.

See also [Ordering of cache and branch predictor maintenance instructions](#).

D4.4.7.3.1 Specific requirements for branch predictor maintenance instructions

If, for a given translation regime and a given VMID as appropriate, the instructions at any virtual address change, then branch predictor maintenance instructions must be performed to invalidate entries in the branch predictor, to ensure that the change is visible to subsequent execution. This maintenance is required when writing new values to instruction locations. It can also be required as a result of any of the following situations that change the translation of a virtual address to a physical address, if, as a result of the change to the translation, the instructions at the virtual addresses change:

- For any translation regime other than the PL1&0 translation regime, enabling or disabling stage 1 translations.
- For the PL1&0 translation regime:
 - When stage 2 translations are enabled, enabling or disabling stage 1 translations unless accompanied by a change of VMID.
 - When stage 2 translations are disabled, enabling or disabling stage 1 translations.
 - Enabling or disabling stage 2 translations.
- Writing new mappings to the [PRBAR](#) or [PRLAR](#) registers unless accompanied by a change to the VMID.

Note

Invalidation is not required if the changes to the translations are such that the instructions associated with the non-faulting translations of a virtual address, for a given translation regime and a given VMID, as appropriate, remain unchanged throughout the sequence of changes to the translations. Examples of translation changes to which this applies are:

- Changing a valid translation to a translation that generates an [Types of MPU faults](#).
 - Changing a translation that generates an [MPU fault](#) to a valid translation.
-

Failure to invalidate entries might give `CONSTRAINED UNPREDICTABLE` results, caused by the execution of old branches. For more information, see [Ordering of cache and branch predictor maintenance instructions](#).

Note

- There is no requirement to use the branch predictor maintenance operations to invalidate the branch predictor after:
 - Changing the ContextID or VMID.
 - A cache maintenance instruction that is identified as also flushing the branch predictors, see [AArch32 cache and branch predictor maintenance instructions](#).
-

[Cache maintenance System instructions](#) shows the branch predictor maintenance operations in a PMSA implementation.

D4.4.7.3.2 Behavior of the branch predictors at reset

In AArch32 state:

- If branch predictors are not architecturally invisible:
 - The branch predictors reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
 - The branch predictors are disabled at reset.
- An implementation can require the use of a specific branch predictor initialization routine to invalidate the branch predictor storage array before it is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, but the routine must be documented clearly as part of the documentation of the device.
- Arm recommends that whenever an invalidation routine is required, it is based on the AArch32 branch predictor maintenance operations.

Similar rules apply to cache behavior, see [Behavior of caches at reset](#).

D4.4.7.4 General requirements for the scope of cache and branch predictor maintenance instructions

The specification of the cache maintenance and branch predictor instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in [General behavior of the caches](#), [Behavior of caches at reset](#), and [Prefetching into cache](#).

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

As a side-effect of a branch predictor maintenance instruction, any entry in the branch predictor might be invalidated.

———— Note ————

Arm recommends that, for best performance, such side-effects are kept to a minimum.

D4.4.7.5 Effects of instructions that operate by VA to the Point of Coherency

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of other PEs in the Shareability domain described by the Shareability attributes of the VA supplied with the instruction.

For Device memory and Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions must affect the caches of all PEs in the Outer Shareable Shareability domain of the PE on which the instruction is operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the Point of Coherency.

Table D4-4 PEs affected by cache maintenance instructions to the Point of Coherency

Shareability	PEs affected	Effective to
Non-shareable	The PE performing the operation	The Point of Coherency of the entire system
Inner Shareable	All PEs in the same Inner Shareable Shareability domain as the PE performing the operation	The Point of Coherency of the entire system
Outer Shareable	All PEs in the same Outer Shareable Shareability domain as the PE performing the operation	The Point of Coherency of the entire system

D4.4.7.6 Effects of instructions that operate by VA but not to the Point of Coherency

The following instruction operate by VA but not to the Point of Coherency:

- Clean data or unified cache line by MVA to the Point of Unification, [DCCMVAU](#).
- Invalidate instruction cache line by MVA to Point of Unification, [ICIMVAU](#).

- Invalidate by MVA from branch predictors, [BPIMVA](#).

For these instructions, [Table D4-5](#) shows how, for a VA in a Normal or Device memory location, the Shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

Table D4-5 PEs affected by cache maintenance instructions to the Point of Unification

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction
Inner Shareable or Outer Shareable	All PEs in the same Inner Shareable Shareability domain as the PE executing the instruction	The Point of Unification of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain as the PE executing the instruction

Note

The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable Shareability domain containing the PE executing the instruction.

D4.4.7.7 Effects of All and set/way maintenance instructions

The [ICIALLU](#), [BPIALL](#) and DC* set/way instructions apply only to the caches and branch predictors of the PE that performs the instruction. If the branch predictors are architecturally-visible, [ICIALLU](#) also performs a [BPIALL](#) operation.

The [ICIALLUIS](#) and [BPIALLIS](#) instructions can affect the caches and branch predictors of all PEs in the same Inner Shareable Shareability domain as the PE that performs the instruction. If the branch predictors are architecturally-visible, [ICIALLUIS](#) also performs a [BPIALLIS](#) operation. These instructions have an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable Shareability domain.

Note

The possible presence of system caches, as described in [System level caches](#), means architecture does not guarantee that all levels of cache can be maintained using set/way instructions. See [Example code for cache maintenance instructions](#) for more information.

D4.4.7.8 Effects of virtualization and security on the AArch32 cache maintenance instructions

Each Security state has its own physical address space, and therefore cache entries are associated with physical address space. In addition, cache maintenance and branch predictor instructions performed have to take account of:

- Whether the instruction was performed at EL1 or at EL2.
- For instructions that operate by VA, the current VMID.

[Table D4-6](#) shows the effects of virtualization and security on these maintenance instructions.

Table D4-6 Effects of virtualization and security on the AArch32 cache maintenance instructions

Cache maintenance instructions	Specified entry
Data or unified cache maintenance instructions	

Cache maintenance instructions	Specified entry
Invalidate, Clean, or Clean and Invalidate by VA: DCIMVAC , DCCMVAC , DCCMVAU , DCCIMVAC	All lines that hold the PA that, in the current translation regime, are mapped to by the combination of all of: - The specified VA. - For an instruction executed at EL1, the current VMID^a. - For an instruction executed at EL0
Invalidate, Clean, or Clean and Invalidate by set/way: DCISW , DCCSW , DCCISW	For an instruction, the line specified by set/way provided that the entry comes from the PA space.
Instruction cache maintenance instructions	
Invalidate by VA: ICIMVAU	All lines corresponding to the specified VA^b in the current translation regime and: For an instruction executed at EL1 or EL0, the current VMID^a.
Invalidate All: ICIALLU , ICIALUIS	Can invalidate any unlocked entry in the instruction cache, and are required to invalidate: - For an instruction executed at EL1, all instruction cache lines containing entries associated with the current VMID. - For an instruction executed at EL2, all instruction lines containing entries.
Branch predictor instructions ^c	
Invalidate by VA: BPIMVA	All lines that, in the current translation regime, are mapped to by the combination of all of: - The specified VA. - For an instruction executed at EL1 or EL0, the current VMID^a.
Invalidate all: BPIALL , BPIALLIS	Can invalidate any unlocked entry in the branch predictor, and are required to invalidate: - For an instruction executed at EL1, all lines containing entries associated with the current VMID. - For an instruction executed at EL2, all lines containing entries.

a. Dependencies on the VMID apply even when the value of [HCR.VM](#) is 0. However, if the PE resets, [VSCTLR.VMID](#) resets to a value that is architecturally UNKNOWN, and the [VSCTLR.VMID](#) field must be set to a known value, that might be zero, as part of the PE initialization sequence.

b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:

- For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
- For a VIPT instruction cache, the cache maintenance applies to entries whose virtual index and physical tag corresponds to the specified address.

For information of types of instruction cache, see Instruction caches.

c. In an implementation where the branch predictors are architecturally invisible, these instructions can execute as NOPs.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in [The interaction of cache lockdown with cache maintenance instructions](#) applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second

virtual machine. To ensure this requirement is met, EL1 invalidate by set/way instructions executed in at EL1 when HCR.VM is 1 and EL2 is enabled can, instead, perform a clean and invalidate by set/way.

- The Data cache invalidate instructions [DCIMVAC](#) and [DCISW](#) perform a cache clean as well as a cache invalidate, meaning [DCIMVAC](#) performs the same invalidation as a [DCCIMVAC](#) instruction, and [DCISW](#) performs the same invalidation as a [DCCISW](#) instruction, if the value of HCR.VM is 1, and the instruction is executed at EL1.
- The Data cache invalidate by set/way instruction [DCISW](#) performs a cache clean as well as a cache invalidate, meaning it performs the same invalidation as a [DCCISW](#) instruction, if the value of HCR.SWIO is 1, and the instruction is executed at EL1.

For more information about the cache maintenance instructions, see [About cache maintenance in AArch32 state](#), [AArch32 cache and branch predictor maintenance instructions](#), and [The Protected Memory System Architecture](#).

D4.4.7.9 Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches regardless of whether the System register Cacheability controls force all memory accesses to be Non-cacheable.

For VA-based cache maintenance instructions, the instructions operate on the caches regardless of the memory type and Cacheability attributes marked for the memory address in the PMSA translation. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
 - Is Normal memory or Device memory.
 - Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

D4.4.7.10 Ordering of cache and branch predictor maintenance instructions

The following rules describe the effect of the memory order model on the cache and branch predictor maintenance instructions:

- All cache and branch predictor maintenance instructions that do not specify an address execute, relative to each other, in program order.
All cache and branch predictor instructions that specify an address:
 - Execute in program order relative to all cache and branch predictor operations that do not specify an address.
 - Execute in program order relative to all cache and branch predictor operations that specify the same address.
 - Can execute in any order relative to cache and branch predictor operations that specify a different address.
- Where a cache maintenance or branch predictor instruction appears in program order before a change to the memory protection regions, the architecture guarantees that the cache or branch predictor maintenance instruction uses the translations that were visible before the change to the memory protection regions.
- Where a change to the memory protection regions appears in program order before a cache maintenance or branch predictor instruction, synchronization is required before performing the cache or branch predictor maintenance instruction, to ensure that the maintenance operation uses the new translations. For more information see [PMSAv8-32 implications for caches](#).
- A [DMB](#) instruction causes the effect of all data or unified cache maintenance instructions appearing in program order before the [DMB](#) to be visible to all explicit memory read and write effects appearing in program order after the [DMB](#).
Also, a [DMB](#) instruction ensures that the effects of any data or unified cache maintenance instruction appearing in program order before the [DMB](#) are observable by any observer in the same required Shareability domain before any data or unified cache maintenance or explicit memory operations appearing in program order after the [DMB](#) are observed by the same observer. Completion of the [DMB](#) does not guarantee the visibility of all data to other observers. For example, all data might not be visible to instruction fetches.
- A [DSB](#) is required to guarantee the completion of all cache maintenance instruction that appear in program order before the [DSB](#) instruction.

- A [Context Synchronization event](#) is required to guarantee the effects of any branch predictor maintenance operation. This means a [Context Synchronization event](#) causes the effect of all completed branch predictor maintenance operations appearing in program order before the [Context Synchronization event](#) to be visible to all instructions after the [Context Synchronization event](#).
This means that, if a branch instruction appears after an invalidate branch predictor operation and before any [Context Synchronization event](#), it is CONSTRAINED UNPREDICTABLE whether the branch instruction is affected by the invalidate. Software must avoid this ordering of instructions, because it might cause CONSTRAINED UNPREDICTABLE behavior.
- Any data or unified cache maintenance instruction by VA must be executed in program order relative to any explicit memory read or write effect on the same PE to an address covered by the VA of the cache instruction if that load or store is to Normal Cacheable memory. The order of memory accesses that result from the cache maintenance instruction, relative to any other memory accesses to Normal Cacheable memory, are subject to the memory ordering rules. For more information, see [Definition of the memory model](#).
Any data or unified cache maintenance instruction by VA can be executed in any order relative to any explicit memory read or write effect on the same PE to an address covered by the VA of the cache maintenance instruction if that load or store is not to Normal Cacheable memory.
- There is no restriction on the ordering of data or unified cache maintenance instruction by VA relative to any explicit memory read or write effect on the same PE where the address of the explicit memory read or write effect is not covered by the VA of the cache instruction. Where the ordering must be restricted, a DMB instruction must be inserted to enforce ordering.
- There is no restriction on the ordering of a data or unified cache maintenance instruction by set/way relative to any explicit memory read or write effect on the same PE. Where the ordering must be restricted, a DMB instruction must be inserted to enforce ordering.
- Software must execute a [Context Synchronization event](#) after the completion of an instruction cache maintenance instruction, to guarantee that the effect of the maintenance instruction is visible to any instruction fetch.

A DSB instruction intended to ensure the completion of cache maintenance instructions or branch predictor instructions must have an access type of both loads and stores.

See also the completion requirements for cache and branch predictor maintenance instructions in [Completion and endpoint ordering](#).

The scope of instruction cache maintenance depends on the type of the instruction cache. For more information, see [Instruction caches](#).

Example D4-1 Cache cleaning operations for self-modifying code

The sequence of cache cleaning operations for a line of self-modifying code on a uniprocessor system is:

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Rt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Rn. Use STRH in the first line
; instead of STR for a 16-bit instruction.
    STR Rt, [Rn]
    DCCMVAU Rn          ; Clean data cache by MVA to point of unification (PoU)
    DSB                 ; Ensure visibility of the data cleaned from cache
    ICIMVAU Rn          ; Invalidate instruction cache by MVA to PoU
    BPIMVA Rn           ; Invalidate branch predictor by MVA to PoU
    DSB                 ; Ensure completion of the invalidations
    ISB                 ; Synchronize the fetched instruction stream
```

D4.4.7.11 Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache Arm strongly recommends that software:

- For data or unified cache maintenance, uses the [CTR.DMinLine](#) value to determine the loop increment size for a loop of data cache maintenance by VA instructions.
- For instruction cache maintenance, uses the [CTR.IMinLine](#) value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

D4.4.7.11.1 Example code for cache maintenance instructions

The cache maintenance instructions by set/way can be used to clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

Note

Because the set/way instructions operate only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Because any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the cache line is not affected by set/way based cache maintenance. Therefore, Arm strongly discourages the use of set/way instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.

The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations with a Normal Cacheable attribute. Such an approach can have very large system performance effects, and Arm advises implementers to use hardware coherency mechanisms in systems where this will be an issue.

[System level caches](#) refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency. It assumes the current Cache Size Identification Register is in 32-bit format.

```

MRC p15, 1, R0, c0, c0, 1 ; Read CLIDR into R0
ANDS R3, R0, #0x07000000
MOV R3, R3, LSR #23 ; Cache level value (naturally aligned)
BEQ Finished
MOV R10, #0

Loop1
ADD R2, R10, R10, LSR #1 ; Work out 3 x cache level
MOV R1, R0, LSR R2 ; bottom 3 bits are the Cache type for this level
AND R1, R1, #7 ; get those 3 bits alone
CMP R1, #2
BLT Skip ; no cache or only instruction cache at this level
MCR p15, 2, R10, c0, c0, 0 ; write CSSELR from R10
ISB ; ISB to sync the change to the CCSIDR
MRC p15, 1, R1, c0, c0, 0 ; read current CCSIDR to R1
AND R2, R1, #7 ; extract the line length field
ADD R2, R2, #4 ; add 4 for the line length offset (log2 16 bytes)
MOV R4, #0x3FF
ANDS R4, R4, R1, LSR #3 ; R4 is the max number on the way size (right aligned)
CLZ R5, R4 ; R5 is the bit position of the way size increment
MOV R9, R4 ; R9 working copy of the max way size (right aligned)

Loop2
MOV R7, #0x00007FFF
ANDS R7, R7, R1, LSR #13 ; R7 is the max number of the index size (right aligned)

Loop3
ORR R11, R10, R9, LSL R5 ; factor in the way number and cache number into R11
ORR R11, R11, R7, LSL R2 ; factor in the index number
MCR p15, 0, R11, c7, c10, 2 ; DCCSW, clean by set/way
SUBS R7, R7, #1 ; decrement the index
BGE Loop3
SUBS R9, R9, #1 ; decrement the way number
BGE Loop2

Skip
ADD R10, R10, #2 ; increment the cache number
CMP R3, R10
DSB ; ensure completion of previous cache maintenance instruction

```

BGT Loop1
Finished

Similar approaches can be used for all cache maintenance instructions.

D4.4.8 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

D4.4.8.1 The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
 - Cache clean by set/way, [DCCSW](#).
 - Cache invalidate by set/way, [DCISW](#).
 - Cache clean and invalidate by set/way, [DCISW](#).
 - Instruction cache invalidate all, [ICIALLU](#) and [ICIALUIS](#).

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the Fault status code defined for this purpose. See [Data Abort exception](#).

This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.

The effect of the following instructions is IMPLEMENTATION DEFINED:

- Cache clean by virtual address, [DCCMVAC](#) and [DCCMVAU](#).
- Cache invalidate by virtual address, [DCIMVAC](#).
- Cache clean and invalidate by virtual address, [DCCIMVAC](#).

However, one of the following approaches must be adopted in all these cases:

1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.
2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the Fault status code defined for this purpose. See [DFSR](#) or [HSR](#).

If [HCR.TIDCP](#) is set to 1, any exception relating to lockdown of an entry is routed to EL2.

Note

An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.

In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.

The reset setting of the cache must be that no cache entries are locked.

D4.4.8.1.1 Additional cache functions for the implementation of lockdown

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED space.

D4.4.9 System level caches

The Arm Architecture defines a *system cache* as a cache that is not described in the PE Cache Identification registers, [CCSIDR](#), and [CLIDR](#), and for which the set/way cache maintenance instructions do not apply.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the point of coherency and cannot be managed by cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the point of coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the Arm architecture is explicitly prohibited.
2. System caches which lie before the point of coherency and can be managed by cache maintenance by address instructions that apply to the point of coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entire system cache must be performed, as is the case for power management, it must be performed using non-architectural mechanisms.
3. System caches which lie beyond the point of coherency and so are invisible to software. The management of such caches is outside the scope of architecture.

D4.5 System register support for IMPLEMENTATION DEFINED memory features

The System register encoding space with $\{\text{coproc} == 0b1111, \text{CRn} == c15\}$ is reserved for IMPLEMENTATION DEFINED registers, and can provide additional registers for the memory system. For more information, see [Organization of registers in the \(coproc == 0b1111 \) encoding space](#).

D4.6 External aborts

The Arm architecture defines External aborts as errors that occur in the memory system, other than those that are detected by the MPU or Debug hardware. An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted, and this corruption is detected by hardware using a parity or error correction code (ECC). The error might have been propagated.

An External abort is one of:

- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see [Exception terminology](#).

The Arm architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous External aborts.

PMSAv8-32 permits External aborts on data accesses, translation table walks, and instruction fetches to be either synchronous or asynchronous. The reported fault code identifies whether the External abort is synchronous or asynchronous.

It is IMPLEMENTATION DEFINED which External aborts, if any, are supported. Asynchronous External aborts generate SError exceptions.

In AArch32 state:

- SError exceptions are taken as asynchronous Data Abort exceptions.
- Synchronous External aborts:
 - On data accesses are taken as synchronous Data Abort exceptions.
 - On instruction fetches, or prefetches, are taken as synchronous Prefetch Abort exceptions.

Normally, External aborts are rare. An imprecise asynchronous External abort is likely to be fatal to the process that is running. Arm recommends that implementations make External aborts precise wherever possible.

The following subsections give more information about possible External aborts:

- [Provision for classification of External aborts](#).
- [Parity or ECC error reporting](#).

The section [Exception reporting in a PMSAv8-32 implementation](#) describes the reporting of External aborts.

D4.6.1 Provision for classification of External aborts

For an External abort taken to a privileged mode other than Hyp mode, an implementation can use the [DFSR.ExT](#) or [IFSR.ExT](#) bits to provide more information about the External abort:

- [DFSR.ExT](#) provides an IMPLEMENTATION DEFINED classification of External aborts on data accesses.
- [IFSR.ExT](#) provides an IMPLEMENTATION DEFINED classification of External aborts on instruction accesses.

For an External abort taken to Hyp mode, the [HSR.EA](#) bit, provides an IMPLEMENTATION DEFINED classification of External aborts.

For all aborts other than External aborts these bits return a value of 0.

D4.6.2 Parity or ECC error reporting

The Arm architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache systems. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors, see [Exception reporting in a PMSAv8-32 implementation](#). However when parity or ECC error reporting is implemented it is IMPLEMENTATION DEFINED whether a parity or ECC error is reported using the assigned fault code, or using another appropriate encoding.

For all purposes other than the Fault status encoding, parity or ECC errors are treated as External aborts.

D4.7 Memory barrier instructions

[Memory barriers](#) describes the memory barrier instructions. This section describes the system level controls of those instructions.

D4.7.1 EL2 control of the maintenance scope of Data Synchronization Barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 and that supports maintenance scope limitations on the [DSB](#) instructions, the [HCR.BSU](#) field can modify the required maintenance scope of an instruction that is executed at EL0 or EL1. [Table D4-7](#) shows the encoding of this field:

Table D4-7 EL2 control of DSB maintenance scope, for constraining both loads and stores executed at EL0 or EL1

HCR.BSU	Minimum maintenance scope of barrier instructions
00	No effect, maintenance scope is as specified by the instruction
01	Inner Shareable
10	Outer Shareable
11	Full system

For an instruction executed at EL0 or EL1, [Table D4-8](#) shows how the [HCR.BSU](#) is combined with the maintenance scope specified by the argument of the [DSB](#) to both loads and stores to give the scope of the instruction.

Table D4-8 Effect of the HCR.BSU on barrier instructions executed at Non-secure EL0 or EL1

Maintenance scope specified by the DSB argument	HCR.BSU	Resultant maintenance scope
Full system	Any	Full system
Outer Shareable	00, 01, or 10	Outer Shareable
	11, Full system	Full system
Inner Shareable	00 or 01	Inner Shareable
	10, Outer Shareable	Outer Shareable
	11, Full system	Full system
Non-shareable	00, No effect	Non-shareable
	01, Inner Shareable	Inner Shareable
	10, Outer Shareable	Outer Shareable
	11, Full system	Full system

D4.8 Pseudocode description of general memory System instructions

This section lists the pseudocode describing general memory operations:

- [Memory data type definitions.](#)
- [Basic memory access.](#)
- [Aligned memory access.](#)
- [Unaligned memory access.](#)
- [Exclusives monitors operations.](#)
- [Access permission checking.](#)
- [Abort exceptions.](#)
- [Memory barriers.](#)

D4.8.1 Memory data type definitions

This section lists the memory data types.

The memory data types are:

- Address descriptor, defined by the [AddressDescriptor](#) type.
- Full address, defined by the [FullAddress](#) type.
- Memory attributes, defined by the [MemoryAttributes](#) type.
- Memory type, defined by the [MemType](#) enumeration.
- Device memory type, defined by the [DeviceType](#) enumeration.
- Normal memory attributes, defined by the [MemAttrHints](#) type.
- Cacheability attributes, defined by the [MemAttr_NC](#), [MemAttr_WT](#), and [MemAttr_WB](#) constants.
- Allocation hints, defined by the [MemHint_No](#), [MemHint_WA](#), [MemHint_RA](#), and [MemHint_RWA](#) constants.
- Access permissions, defined by the [Permissions](#) type.

D4.8.2 Basic memory access

The [PhysMemRead\(\)](#) and [PhysMemWrite\(\)](#) functions perform single-copy atomic, aligned, little-endian memory accesses of `size` bytes to or from the underlying physical memory array of bytes.

The attributes in `memaddrdesc.memattrs` are used by the memory system to determine caching and ordering behaviors as described in [Memory types and attributes](#), [Definition of the memory model](#), and [Atomicity in the Arm architecture](#).

D4.8.3 Aligned memory access

The [AArch32_MemSingle\(\)](#) functions make atomic, little-endian accesses of `size` bytes.

D4.8.4 Unaligned memory access

See [Unaligned data access](#) for details of the [SCTLR.A](#) and [HSCTLR.A](#) controls on the generation of alignment faults. The [HSCTLR](#) control applies to Normal memory accesses from Hyp mode, and the [SCTLR](#) control applies to Normal memory accesses from all other modes.

The [Mem_with_type\(\)](#) functions make an access of the required type. If that access is naturally aligned, each form of the function performs an atomic access by making a single call to [AArch32_MemSingle\(\)](#). If that access is not aligned but passes the [IsAlignedSize\(\)](#) checks, each form of the function synthesizes the required access from multiple calls to [AArch32_MemSingle\(\)](#). It also reverses the byte order if the access is big-endian.

D4.8.5 Exclusives monitors operations

The `AArch32_SetExclusiveMonitors()` function sets the Exclusives monitors for a Load-Exclusive instruction, for a block of bytes. The size of the blocks is determined by `size`, at the VA address. The `AArch32_ExclusiveMonitorsPass()` function checks whether a Store-Exclusive instruction still has possession of the Exclusives monitors and therefore completes successfully.

The `AArch32_ExclusiveMonitorsPass()` function checks whether a Store-Exclusive instruction still has possession of the Exclusives monitors, by checking whether the Exclusives monitors are set to include the location of the memory block specified by `size`, at the virtual address defined by `address`. The atomic write that follows after the Exclusives monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to `AArch32_SetExclusiveMonitors()`.

The `ExclusiveMonitorsStatus()` function returns 0 if the previous atomic write was to the same physical memory locations selected by `AArch32_ExclusiveMonitorsPass()` and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

The `MarkExclusiveGlobal()` procedure takes as arguments a `FullAddress.address`, the PE identifier `processorid` and the size of the transfer. The procedure records that the PE `processorid` has requested exclusive access covering at least `size` bytes from address `address`. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is CONSTRAINED UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

The `MarkExclusiveLocal()` procedure takes as arguments a `FullAddress.address`, the PE identifier `processorid` and the size of the transfer. The procedure records in a local record that PE `processorid` has requested exclusive access to an address covering at least `size` bytes from address `address`. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

The `IsExclusiveGlobal()` function takes as arguments a `FullAddress.address`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked in a global record an address range as exclusive access requested that covers at least `size` bytes from address `address`. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE.

The `IsExclusiveLocal()` function takes as arguments a `FullAddress.address`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked an address range as exclusive access requested that covers at least the `size` bytes from address `address`. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of `size` bytes from address `address`. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

The `ClearExclusiveByAddress()` procedure takes as arguments a `FullAddress.address`, the PE identifier `processorid` and the size of the transfer. The procedure clears the global records of all PEs, other than `processorid`, for which an address region including any of `size` bytes starting from `address` has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the PE `processorid` is also cleared if any of `size` bytes starting from `address` has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

The `ClearExclusiveLocal()` procedure takes as arguments the PE identifier `processorid`. The procedure clears the local record of PE `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE `processorid` that an address has had a request for an exclusive access.

D4.8.6 Access permission checking

The `AArch32_S1CheckPermissions()` and `AArch32_S2CheckPermissions()` functions are used by the architecture to perform access permission checking based on attributes derived from the memory accesses presented to the MPUs.

The interpretation of access permission is shown in Memory access control.

D4.8.7 Abort exceptions

The function `AArch32_Abort()` generates a Data Abort exception or a Prefetch Abort exception by calling the `AArch32_TakeDataAbortException()` or `AArch32_TakePrefetchAbortException()` function.

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault.

The function `NoFault()` returns a null record that indicates no fault. The `IsFault()` function tests whether a `FaultRecord` contains a fault.

D4.8.8 Memory barriers

The definition for the memory barrier functions is given by the enumerations `MBMaintenanceScope` and `MBReqTypes`.

These enumerations define the required Shareability domains and required access types used as arguments for DMB and DSB instructions.

The procedures `DataMemoryBarrier()`, `DataSynchronizationBarrier()`, and `InstructionSynchronizationBarrier()` perform the memory barriers.

Chapter D5

The AArch32 Protected Memory System Architecture

This chapter provides a system-level view of the memory system architecture for an Armv8-R implementation, the *Protected Memory System Architecture* (PMSAv8-32). It contains the following sections:

- [About the AArch32 Protected Memory System Architecture](#)
- [Memory protection units](#)
- [Default memory maps and Background region checks](#)
- [Protection region attributes and access permissions](#)
- [PMSAv8-32 memory aborts](#)
- [Exception reporting in a PMSAv8-32 implementation](#)
- [PMSAv8-32 implications for caches](#)

D5.1 About the AArch32 Protected Memory System Architecture

IHXVHB	The Armv8-R AArch32 architecture supports the Protected Memory System Architecture (PMSAv8-32) at EL1 and EL2. PMSAv8-32 is not backwards-compatible with PMSAv7 or earlier Arm PMSAs. For general information about the Arm memory model, see: • The AArch32 Application Level Memory Model • The AArch32 System Level Memory Model
RCHCTW	The PMSAv8-32 uses a flat mapping between the <i>virtual address</i> (VA) accessed by the PE and the 32-bit <i>physical address</i> (PA) accessed in the memory system. That is, for all accesses, the VA is the same as the PA, see Address translation and translation regimes in PMSAv8-32.
RQKPHH	PMSAv8-32 only supports a unified memory protection scheme. It does not support separate instruction and data regions in the address map.

D5.1.1 Protection regions

RDZDNX	An MPU defines protection regions in the 4GB address map.
RTXYCY	The definition of a protection region specifies the start and the end of the region, the access permissions to the region, and the memory attributes for the region. An individual protection region is a contiguous memory region for which the MPU defines the memory attributes, for example the memory type and the access permissions. An individual protection region: • Is defined by a pair of a Base Address Register and a Limit Address Register, see Memory protection units (MPUs). • Has a minimum size of 64 bytes. • Has a maximum size of the entire address map, 4GB.
RJQBZH	When multiple protection regions are defined, the protection regions: • Must not overlap. • Do not need to be contiguous. For more information, see Memory protection.
RBWKMx	PMSAv8-32 defines two MPUs: EL1 MPU Defines the protection regions for accesses from EL1 and from EL0. EL2 MPU Defines the protection regions for accesses from EL2. When the value of HCR.VM is 1 and for accesses from EL1 and from EL0, the EL2 MPU uses these protection regions to modify the access permissions and memory attributes that are assigned by the EL1 MPU. For an address that does not match any defined protection region, PMSAv8-32 defines Background regions and default memory maps which are used by both the EL1 and EL2 MPU. For more information, see: • Memory protection units • Protection region attributes and access permissions. • Default memory maps and Background region checks.

D5.1.2 The Protected Memory System Architecture and Virtualization

RSTTXM	A PMSAv8-32 implementation can provide a virtualization scheme in which a single PE supports multiple guest environments under the control of a single hypervisor that executes at EL2, where executing at EL2 means executing in Hyp mode.
IQTNUM	Typically, a realtime operating system executing at EL1 programs the EL1 MPU to define the memory map for its own operation and for applications it runs at EL0. A hypervisor executing at EL2 programs the EL2 MPU to define the memory map for its own operation and to modify the access permission and memory attribute assignments that are performed by the EL1&0 stage 1 translation.

D5.2 Memory protection units

RFGXJQ An MPU checks whether the address used by a memory access matches a defined protection region. It uses the permissions and memory attributes that are defined for that region or for the Background region to determine whether the access is permitted and if it is permitted, how it must behave.

- The EL1 MPU is enabled when the value of **SCTLR.M** is 1.
- The EL2 MPU is enabled when the value of **HSCTLR.M** is 1.

RTKKKV Each MPU provides two mechanisms for accessing memory region attributes:

- Using indirect accesses, where:
 - **PRSEL**/**HPRSEL** is programmed to specify the required protection region.
 - **PRBAR**/**HPRBAR** and **PRLAR**/**HPRLAR** are programmed to specify the address range and attributes for that region.
- Directly specifying the address range and attributes for the first 32 regions using **PRBAR**<n>/**HPRBAR**<n> and **PRLAR**<n>/**HPRLAR**<n>.

For the EL2 protection regions, **HPREN** provides direct access to the region enable for regions 0-31.

D5.2.1 Effect of enabling one or both MPUs on attribute assignment and fault generation

RGQHFQ For accesses that fault:

- All MPU faults are treated as level 0 faults. For translation operations, PMSA contexts return the 64-bit PAR format. The PAR format discards the least significant bits but the translation query uses the entire input address. For more information, see the [Types of MPU faults](#). For fault status codes, see **DFSR**, **IFSR**, and **HSR**.
- If the value of **EDSCR.HDD** is 1, stage 2 faults in Debug state are reported as Debug events at EL1. For more information, see Behavior in Debug state.

RKTJZT For an access that does not fault:

When both the EL1 MPU and the EL2 MPU are enabled:

- For an EL2 access (an access from Hyp mode), the EL2 MPU configuration settings determine the access permissions and memory attributes.
- For EL1 and EL0 accesses, the EL1 MPU configuration settings determine the stage 1 access permissions and memory attributes. If the *Effective value* of **HCR.VM** is 1, those access permissions and memory attributes are then modified by the EL2 MPU, as described in [Combining attributes](#).

When the EL1 MPU is disabled and the EL2 MPU is enabled:

- For an EL2 access (an access from Hyp mode), the EL2 MPU configuration settings determine the access permissions and memory attributes.
- For an EL1 or EL0 access, the EL1 Background region settings determine the stage 1 access permissions and memory attributes. If the *Effective value* of **HCR.VM** is 1, those access permissions and memory attributes are then modified by the EL2 MPU, as described in [Combining attributes](#).

When the EL1 MPU is enabled but the EL2 MPU is disabled:

- For an EL2 access (an access from Hyp mode), the EL2 Background region settings determine the access permissions and memory attributes.
- For EL1 and EL0 accesses, the EL1 MPU configuration settings determine the access permissions and memory attributes. If the *Effective value* of **HCR.VM** is 1, those access permissions and memory attributes are then combined with the EL2 Background region settings, as described in [Combining attributes](#).

When both the EL1 MPU and the EL2 MPU are disabled:

- For an EL2 access (an access from Hyp mode), the EL2 Background region settings determine the stage 1 access permissions and memory attributes.
- For an EL1 or EL0 access, the EL1 Background region settings determine the access permissions and memory attributes. If the *Effective value* of **HCR.VM** is 1, those access permissions and memory attributes are then modified by the EL2 Background region settings, as described in [Combining attributes](#).

IPTVNN Combining attributes is trivial if the EL1 MPU check resulted in attributes from the EL1 Background region because the two Background regions are identical when both of them are enabled.

For more information on the fault checking sequence, see [The MPU fault-checking sequence](#).

D5.2.2 Address translation and translation regimes in PMSAv8-32

IJBGJ

In the Armv8-R AArch 32 architecture:

- Address translation describes the process of flat mapping the *virtual address* (VA) used by the PE to the *physical address* (PA) accessed in the memory system and determining the access permissions and memory attributes of the target PA.
- A *translation regime* maps a VA to a PA, using one or two stages of address translation to assign the access permissions and memory attributes of the target PA. When two translation stages are used, the intermediate address is treated as an *intermediate physical address* (IPA).

Note

These definitions provide consistency between the Armv8-A and Armv8-R descriptions of the memory model.

RVSJC

PMSAv8-32 defines the following translation regimes:

EL1&0 translation regime

This assigns the access permissions and memory attributes for any access from EL1 or EL0.

This translation regime has one or two stages of translation:

- All accesses from EL1 or EL0 are translated by the EL1 MPU if it is enabled. This translation is a stage 1 translation.
- When the value of [HCR.VM](#) is 1, the access is also translated by the EL2 MPU. This translation is a stage 2 translation, and can modify the access permissions and memory attributes that are assigned by the stage 1 translation.

For the EL1&0 stage 1 translation, ADDRESS is in protection region *n* if:

[PRBAR<n>.BASE::'000000'](#) <= ADDRESS <= [PRLAR<n>.LIMIT::'111111'](#)

EL2 translation regime

This assigns the access permissions and memory attributes for any access from EL2.

This translation regime has a single stage of translation, stage 1, that is performed by the EL2 MPU.

For the EL2 translation, ADDRESS is in protection region *n* if:

[HPRBAR<n>.BASE::'000000'](#) <= ADDRESS <= [HPRLAR<n>.LIMIT::'111111'](#)

RVFSXY

The attributes for a protection region are defined by the combination of:

- The values that are programmed into the Base Address Register and Limit Address Register pair that define the protection region.
- A Memory Attributes Indirection register that is indexed by those values.

[Figure D5-1](#) shows how memory accesses are handled when both MPUs are enabled.

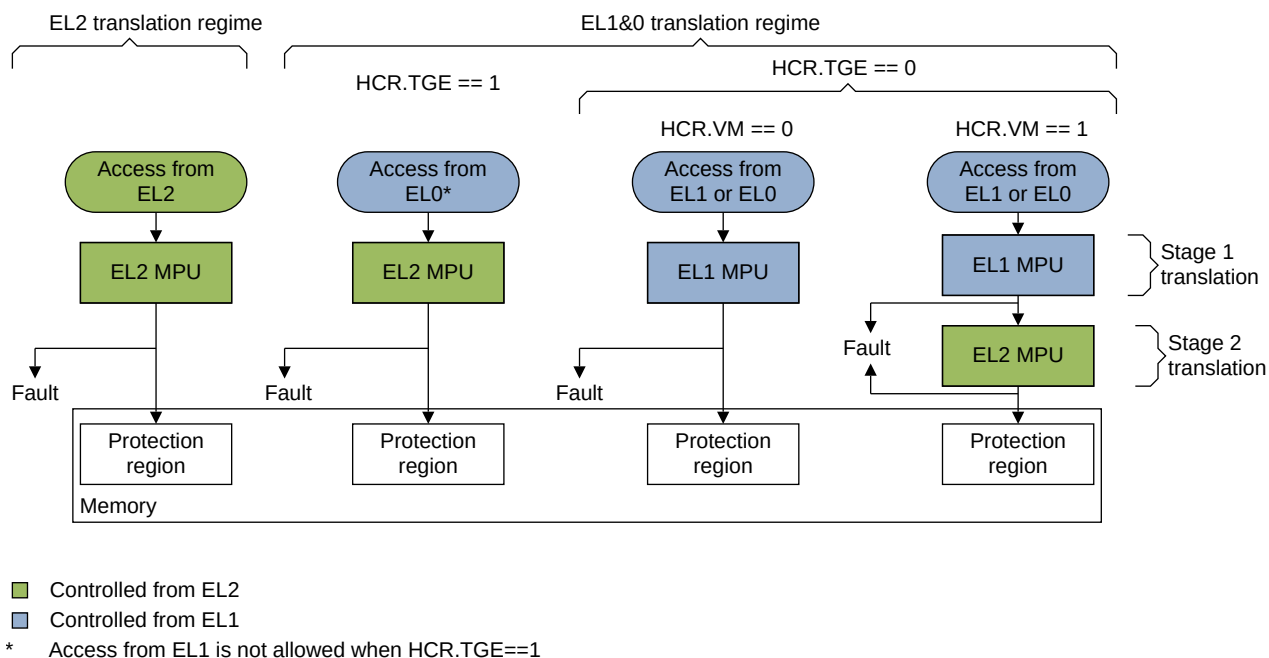


Figure D5-1 PMSAv8-32 memory access permission and attribute control

For other scenarios, see [Effect of enabling one or both MPUs on attribute assignment and fault generation](#).

D5.3 Default memory maps and Background region checks

- RCYDZW** Each PMSAv8-32 MPU has an associated default memory map which is used when the MPU is not enabled.
- RKDMFP** When the MPU is enabled and Background region checking is enabled, privileged accesses that do not hit defined protection regions undergo a second check.
- For the EL1 MPU, Background region checking is enabled for privileged access when the value of **SCTLR.BR** is 1.
 - For the EL2 MPU, Background region checking is enabled for accesses from EL2 when the value of **HSCTLR.BR** is 1.
- For details, see [Effect of enabling one or both MPUs on attribute assignment and fault generation](#).

D5.3.1 EL1 MPU default memory map

- RTFFQW** [Table D5-1](#) and [Table D5-2](#) describe the default memory map defined for the EL1 MPU. When the value of **HCR.DC** is 0, Background region checking by the EL1 MPU also uses this memory map.

Table D5-1 Instruction accesses and EL1 Background region checks

Address range	SCTLR.I == 0	SCTLR.I == 1	XN
0x00000000 - 0x7FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Non-shareable, Write-Through Cacheable	Execution permitted
0x80000000 - 0xFFFFFFFF	Not applicable	Not applicable	Execute-never

Table D5-2 Data accesses and EL1 Background region checks

Address range	SCTLR.C == 0	SCTLR.C == 1
0x00000000 - 0x3FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Non-shareable, Write-Back, Write-Allocate Cacheable
0x40000000 - 0x5FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Non-shareable, Write-Through Cacheable
0x60000000 - 0x7FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Shareable, Non-cacheable
0x80000000 - 0xBFFFFFFF	Device-nGnRE	Device-nGnRE
0xC0000000 - 0xFFFFFFFF	Device-nGnRnE	Device-nGnRnE

- RZQKBH** When the value of **HCR.DC** is 1, the Background region and the Default memory map produce accesses with the following attributes:
- Normal memory.
 - Non-shareable.
 - Inner Write-Back Read-Allocate, Write-Allocate Cacheable.
 - Outer Write-Back Read-Allocate, Write-Allocate Cacheable.
 - Execution permitted.

D5.3.2 EL2 MPU default memory map

- RGZHVJ** [Table D5-3](#) and [Table D5-4](#) describe the default memory map defined for the EL2 MPU. Background region checking by the EL2 MPU also uses this memory map.

Table D5-3 Instruction accesses and EL2 Background region checks

Address range	HSCTLR.I == 0	HSCTLR.I == 1	XN
0x00000000 - 0x7FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Non-shareable, Write-Through Cacheable	Execution permitted
0x80000000 - 0xFFFFFFFF	Not applicable	Not applicable	Execute-never

Table D5-4 Data accesses and EL2 Background region checks

Address range	HSCTLR.C == 1	HSCTLR.C == 1
0x00000000 - 0x3FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Non-shareable, Write-Back, Write-Allocate Cacheable
0x40000000 - 0x5FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Non-shareable, Write-Through Cacheable
0x60000000 - 0x7FFFFFFF	Normal, Shareable, Non-cacheable	Normal, Shareable, Non-cacheable
0x80000000 - 0xBFFFFFFF	Device-nGnRE	Device-nGnRE
0xC0000000 - 0xFFFFFFFF	Device-nGnRnE	Device-nGnRnE

D5.4 Protection region attributes and access permissions

IRKYYM	The protection region attribute fields control the memory type, Cacheability, and Shareability of the region. Armv8-R uses the same memory types and memory attributes as Armv8-A.
	Note The Armv8-R identification scheme is based on the scheme that is used by VMSAv8-32 when using the Long-descriptor translation table format.
RVLDLX	The scheme determines how the descriptions of the properties are grouped. The scheme is based on the following model: Access controls Determine whether the PE can access the output address that corresponds to the given input address. If not, an MPU fault is generated and there is no memory access. Attributes Are valid only for an output address that the PE, in its current state, can access. The attributes define aspects of the required behavior of accesses to the target memory region. The memory attributes and access permissions for a protection region are defined by entries in: • The PRBAR and PRLAR, or HPRBAR and HPRLAR, that define the region. • The MAIR<<i>m</i>>.Attr<<i>n</i>> or HMAIR<<i>m</i>>.Attr<<i>n</i>> field that is indexed by PRLAR.AttrIdx or HPRLAR.AttrIdx, respectively.
RJDLFX	If the memory attributes for an enabled protection region or the background region do not match the attributes defined by the memory access the relevant MPU generates a fault. See also Assignment model of memory attributes and access permissions.
RRMHGK	For the EL1&0 translation regime, when both MPUs are enabled and the value of HCR.VM is 1, the stage 1 memory attribute and access permission assignments are combined with the stage 2 assignments as described in Combining attributes.

D5.4.1 Assignment model of memory attributes and access permissions

IHBXLF	PRBAR.SH; HPRBAR.SH Defines the Shareability, for a Normal memory protection region. For any type of Device memory, and for Normal Inner Non-cacheable, Outer Non-cacheable memory, the value of the SH[1:0] field is IGNORED. PRBAR.AP[2:1]; HPRBAR.AP[2:1] Defines the Access permissions. For encoding information, see the register description. PRBAR.XN; HPRBAR.XN Defines the Execute-never attribute for the region. For encoding information, see the register description. XN == 1 The region is Execute-never. PMSAv8-32 makes no distinction between Privileged execute-never (PXN) and Unprivileged execute-never (UXN). PRLAR.AttrIdx; HPRLAR.AttrIdx Indexes an Attr<<i>n</i>> field in MAIR<<i>m</i>> or HMAIR<<i>m</i>>, as follows: • AttrIdx[2] indicates the MAIR register to be used: AttrIdx[2] == 0 Use MAIR0 or HMAIR0. AttrIdx[2] == 1 Use MAIR1 or HMAIR1. • AttrIdx[2:0] indicates the required Attr field, Attr<<i>n</i>>, where n = AttrIdx[2:0]. MAIR<<i>m</i>>.Attr<<i>n</i>>; HMAIR< <i>m</i>>.Attr<<i>n</i>> Defines: • The memory type, Normal memory, or a type of Device memory. • For Normal memory: — The Inner Cacheability and Outer Cacheability attributes, each of which is one of Non-cacheable, Write-Through Cacheable, or Write-Back Cacheable. — For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate policy hints, each of which is Allocate or Do not allocate.
--------	---

D5.4.2 Access permissions for instruction execution

RSICLL	Execute-never controls provide an additional level of control on memory accesses permitted by the access permissions settings. These controls are: XN, Execute-never PRBAR.XN and HPRBAR.XN, apply to execution at any Exception level to which the protection region applies. A value of 0 indicates that this control permits execution.
RQZKVB	Executing an instruction at ELx generates a Permission fault unless all of the following are true for the instruction address: - Any stage 1 execute-never control that applies to execution at ELx permits execution. - If the translation regime that applies to ELx has two stages of translations, the stage 2 execute-never control that applies to execution at ELx permits execution. - Read access is permitted.
IVZWDV	In addition, System register controls can enforce execute-never restrictions, regardless of the settings in the XN fields, see Preventing execution from writable locations.
RDTVJH	The execute-never controls apply also to speculative instruction fetching. This means a speculative instruction fetch from a memory region that is execute-never at the current level of privilege is prohibited. The execute-never controls means that, when the stage of address translation is enabled, the PE can fetch, or speculatively fetch, an instruction from a memory location only if all of the following apply: - In the Protection Region Base Address Register for the location the following apply: - The stage 1 execute-never control for the Exception level at which the instruction is executed permits execution. - For a translation regime with two stages of address translation, the stage 2 execute-never control that applies to the Exception level at which the instruction is executed permits execution. - The access permissions permit a read access from the current PE mode. - No other Prefetch Abort condition exists. - Although the XN control applies to speculative fetching, on a speculative instruction fetch from an XN location, no Permission fault is generated unless the PE attempts to execute the instruction that would have been fetched from that location. This means that, if a speculative fetch from an XN location is attempted, but there is no attempt to execute the corresponding instruction, a Permission fault is not generated. - The software that defines a protection region must mark any region of memory that is read-sensitive as XN, to avoid the possibility of a speculative fetch accessing the memory region. This means it must mark any memory region that corresponds to a read-sensitive peripheral as XN. Hardware does not prevent speculative accesses to a region of any Device memory type unless that region is also marked as execute-never for all Exception levels from which it can be accessed.
RXVPBN	When no stage of address translation for the translation regime is enabled, memory regions cannot have XN attributes assigned. Effect of enabling one or both MPUs on attribute assignment and fault generation describes how disabling all MPUs affects instruction fetching.

D5.4.2.1 Preventing execution from writable locations

RGFEND	The architecture provides control bits that, when the corresponding stage 1 address translation is enabled, force writable memory to be treated as XN, regardless of the value of the XN field. When the translation stages are controlled by an Exception level that is using AArch32: - For PL1&0 stage 1 translations, when SCTLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN. - For PL1&0 stage 1 translations, when SCTLR.UWXN is set to 1, an instruction fetch is treated as accessing a PXN region if it accesses a region that software executing at EL0 can write to. - For EL2 stage 1 translations, when HSCCLR.WXN is set to 1, all regions that are writable at stage 1 of the address translation are treated as XN. Note - The SCTLR.WXN controls are intended to be used in systems with very high security requirements. - Setting a WXN or UWXN bit to 1 changes the interpretation of the attributes for the memory protection region, overriding a zero value of an XN field. It does not cause any change to the attributes for the memory protection region.
--------	--

ITJJWS For any given virtual machine, Arm expects WXN and UWXN to remain static in normal operation. A generic sequence to ensure synchronization of a change to these fields, when that change is made without a corresponding change of VMID, is:

```
Change the WXN or UWXN bit.  
  
ISB; // This ensures synchronization of the change.
```

For more information about address translation, see [Address translation and translation regimes in PMSAv8-32](#).

D5.4.3 Combining attributes

RFBPVX For the EL1&0 translation regime, when both MPUs are enabled and the value of HCR.VM is 1, the stage 1 and stage 2 attributes and access permissions are combined as follows:

- If the stage 1 access permissions indicate that an access is not permitted, a stage 1 Permission fault is generated regardless of the stage 2 permissions.
- If the stage 1 access permissions indicate that an access is permitted but the stage 2 access permissions indicate that it is not permitted, a stage 2 Permission fault is generated.
- If an access is permitted by both the stage 1 and the stage 2 access permissions, the memory attributes are combined as shown in [Table D5-5](#), [Table D5-6](#), and [Table D5-7](#).

Table D5-5 Combining the stage 1 and stage 2 memory type assignments

Rule	If the memory type assigned by either stage is:	The resultant memory type is:
Device has precedence over Normal	Any Device memory type	A Device memory type
non-Gathering has precedence over Gathering	A Device-nGxx memory type	A Device-nGxx memory type
non-Reordering has precedence over Reordering	A Device-nGnRx memory type	A Device-nGnRx memory type
No Early Write Acknowledgement has precedence over Early Write Acknowledgement	The Device-nGnRnE memory type	The Device-nGnRnE memory type

RFDGNX Regardless of any Shareability attribute that results from the combinations that are described in [Table D5-7](#):

- Any location for which the resultant memory type is any type of Device memory is always treated as Outer Shareable.
- Any location for which the resultant memory type is Normal Inner Non-cacheable, Outer Non-cacheable is always treated as Outer Shareable.

[Table D5-6](#) shows the assignments at each stage of translation are combined to create the resultant Cacheability attribute. These rules apply independently for the Inner Cacheability and Outer Cacheability attributes.

Table D5-6 Combining Cacheability assignments for Normal memory

Stage 1 assignment	Stage 2 assignment	Resultant Cacheability
Non-cacheable	Any	Non-cacheable
Any	Non-cacheable	Non-cacheable
Write-Through Cacheable	Write-Through or Write-Back Cacheable	Write-Through Cacheable

Stage 1 assignment	Stage 2 assignment	Resultant Cacheability
Write-Through or Write-Back Cacheable	Write-Through Cacheable	Write-Through Cacheable
Write-Back Cacheable	Write-Back Cacheable	Write-Back Cacheable

R_{HYQDR}

A protection region is treated as Outer Shareable, regardless of any Shareability assignments at either stage of translation if:

- The resultant memory type attribute, as described in [Table D5-5](#), is any type of Device memory.
- The resultant memory type attribute, as described in [Table D5-5](#), is Normal memory, and the resultant Cacheability, as described in [Table D5-6](#), is Inner Non-cacheable, Outer Non-cacheable.

For a protection region with a resultant memory type attribute of Normal that is not Inner Non-cacheable, Outer Non-cacheable, [Table D5-7](#) shows how the Shareability attribute is obtained from the assignment at each stage of translation.

Table D5-7 Combining Shareability assignments for Normal not Inner Non-cacheable, Outer Non-cacheable memory

Stage 1 assignment	Stage 2 assignment	Resultant Shareability
Outer Shareable	Any	Outer Shareable
Inner Shareable	Outer Shareable	Outer Shareable
Inner Shareable	Inner Shareable	Inner Shareable
Inner Shareable	Non-shareable	Inner Shareable
Non-shareable	Outer Shareable	Outer Shareable
Non-shareable	Inner Shareable	Inner Shareable
Non-shareable	Non-shareable	Non-shareable

D5.5 PMSAv8-32 memory aborts

IYPWPQ	In a PMSAv8-32 implementation, the following mechanisms cause a PE to take an exception on a failed memory access: Debug exception An exception caused by the debug configuration, see AArch32 Self-hosted Debug. Alignment fault An Alignment fault is generated if the address used for a memory access does not have the required alignment for the operation. For more information, see Unaligned data access and Alignment faults. MPU fault An MPU fault is a fault generated by the fault checking sequence for the current translation regime. See Types of MPU faults. External abort Any memory system fault other than a Debug exception, an Alignment fault, or an MPU fault. Collectively, these mechanisms are called <i>aborts</i>. AArch32 Self-hosted Debug and Halting Debug Events describe Debug exceptions, and the remainder of this section describes Alignment faults, MPU faults, and External aborts.
RNQXHQ	An access that causes an abort is said to be aborted, and uses the <i>Fault Address Registers</i> (FARs) and <i>Fault Status Registers</i> (FSRs) or <i>Exception Syndrome Registers</i> (ESRs) to record context information. The exception generated on a synchronous memory abort: - On an instruction fetch is called the Prefetch Abort exception. - On a data access is called the Data Abort exception. Note The Prefetch Abort exception applies to any synchronous memory abort on an instruction fetch. It is not restricted to speculative instruction fetches.
RNZWDF	The Exception level and PE mode that a PMSAv8-32 memory abort is taken to depends on the translation regime and stage that generate the abort. The fault context is dependent on whether: - The abort is reported as a Prefetch Abort or as a Data Abort. - The exception is taken from the same or a lower Exception level. For more information, see Routing of aborts taken to AArch32 state.
RXQPPG	External aborts can be reported synchronously or asynchronously. Asynchronous External aborts are reported using the SError exception. For more information, see External aborts.
RFTHTQD	In AArch32 state, asynchronous memory aborts are a type of External abort, and are treated as a type of Data Abort exception.
IKDMIQ	The following sections describe the abort mechanisms: Types of MPU faults. The MPU fault-checking sequence. Alignment faults. AArch32 state prioritization of synchronous aborts from a single stage of address translation. An access that causes an abort is said to be aborted. On an abort, System registers are used to record context information. For more information, see Exception reporting in a PMSAv8-32 implementation.

D5.5.1 Types of MPU faults

ISJCRL	This section describes the faults that might be detected during one of the fault-checking sequences described in The MPU fault-checking sequence. MPU faults are always synchronous. When an MPU fault generates an abort for a region of memory, no memory access is made if that region is or could be marked as any type of Device memory. The MPU faults that might be detected during a fault checking sequence are: Permission fault. Translation fault.
--------	---

D5.5.1.1 Permission fault

IDWMXC	A Permission fault can be generated if the memory access does not meet the permissions defined for the Memory protection region. The access type can be Read, Write or Execute.
RMJFXD	Cache or branch predictor maintenance operations cannot cause a Permission fault.

D5.5.1.2 Translation fault

RWNSJK	A Translation fault is generated if the input address for a translation either does not map onto an address range of a Protection region , or the Protection region range that it maps onto is disabled. In these cases the fault is reported as a level 1 Translation fault on the translation stage at which the mapping to a region described by a Protection region failed.
RJTWYB	A data or unified cache maintenance by VA instruction can generate a Translation fault. However: • If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a data or unified cache maintenance by VA to the Point of Coherency instruction can generate a Translation fault. • If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Translation fault. It is IMPLEMENTATION DEFINED whether an instruction cache invalidate by VA operation can generate a Translation fault. It is IMPLEMENTATION DEFINED whether a branch predictor maintenance operation can generate a Translation fault.

D5.5.2 The MPU fault-checking sequence

IMVKHL	This section describes the MPU checks made for the memory accesses required for instruction fetches and for explicit memory effects: • If an instruction fetch faults it generates a Prefetch Abort exception. • If an data memory access faults it generates a Data Abort exception. For more information about Prefetch Abort exceptions and Data Abort exceptions see Handling exceptions that are taken to an Exception level using AArch32.
RJQSLC	In PMSAv8-32, all memory accesses require VA to PA translation. Therefore, when a corresponding stage of address translation is enabled, each access requires a lookup of the attributes of the memory protection region for the accessed VA.
RBQZPH	MPU fault checking is performed for each stage of address translation.
IPQWBM	The fault-checking sequence shows a translation from an Input address to an Output address. For more information about this terminology, see Address translation and translation regimes in PMSAv8-32 .
IKWNJK	Types of MPU faults describes the faults that an MPU fault-checking sequence can report.
RKRLBY	Figure D5-2 shows the stage 1 translation for EL1 and EL0 accesses.

Note

[Figure D5-2](#) and [Figure D5-3](#) give an overview of the fault checking performed by the MPU. See [AArch32 state prioritization of synchronous aborts from a single stage of address translation](#) for the complete set of possible faults and their prioritization.

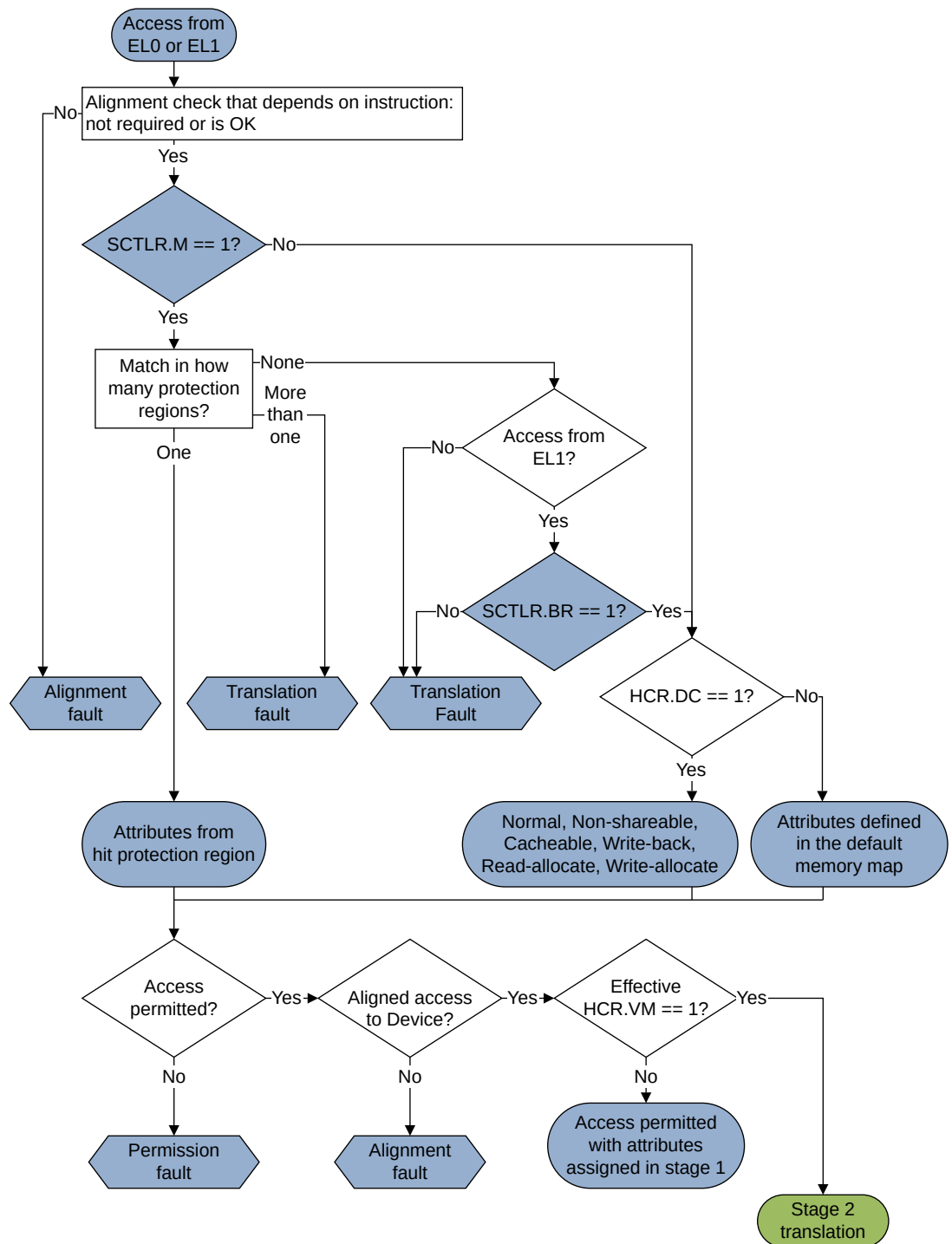


Figure D5-2 Stage 1 translation for EL1 and EL0 accesses

Figure D5-3 shows the stage 2 translation for EL1 and EL0 and stage 1 translation for EL2 accesses.

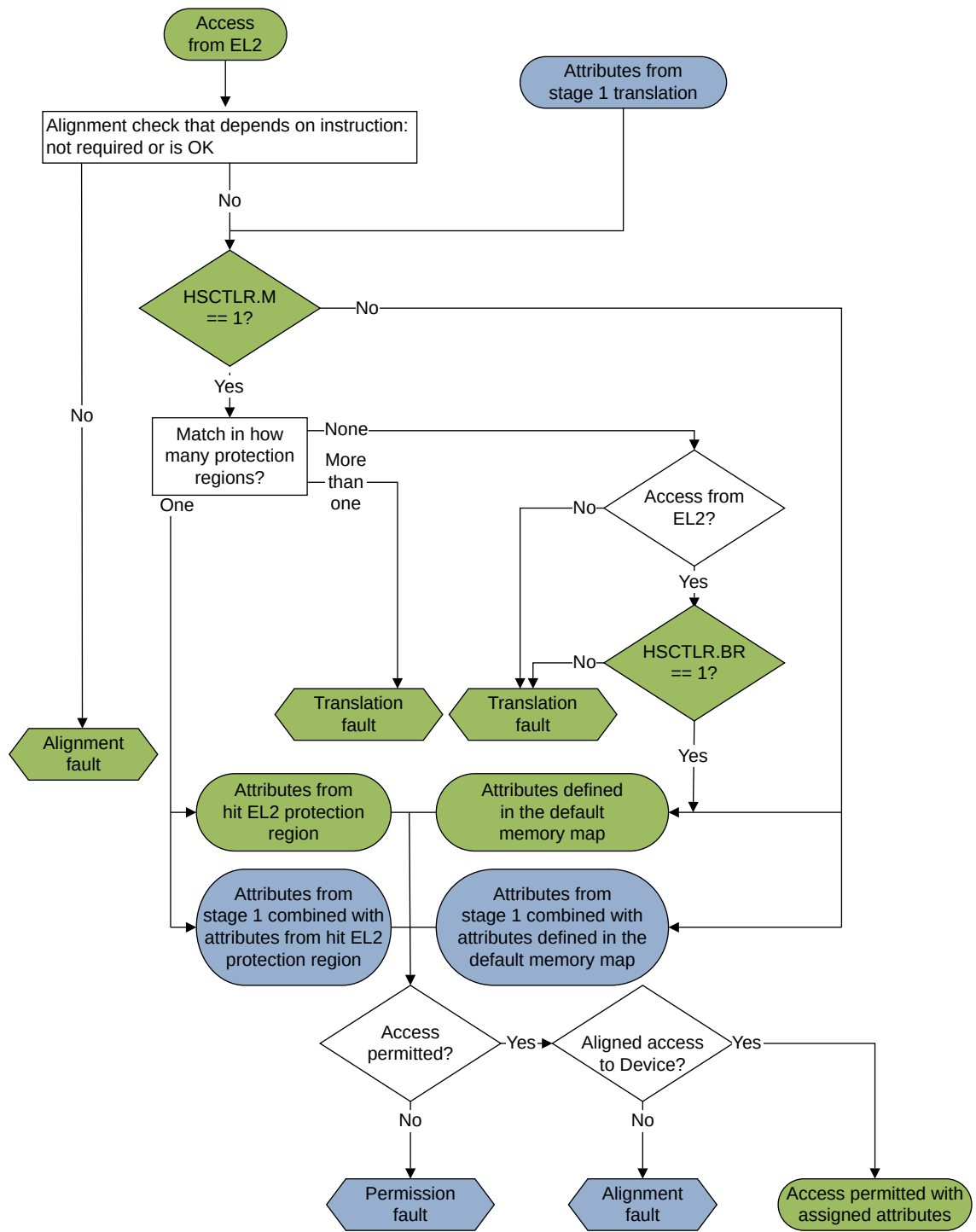


Figure D5-3 Stage 2 translation for EL1 and EL0, stage 1 translation for EL2 accesses

D5.5.2.1 EL1&0 Transaltion

R _{XXGXT}	When EL2 is operating in a PL1 or EL0 mode, any memory access goes through two stages of translation: • Stage 1, from VA to IPA. • Stage 2, from IPA to PA.
R _{DBFDD}	On performing a stage 1 translation, the input addresses must be translated from IPA to PA, using a stage 2 translation. This means that a memory access made as part of a stage 1 translation lookup might generate, on a stage 2 translation: • A Permission fault. • A synchronous External abort on the memory access. HSR.ISS[7] is set to 1, to indicate a stage 2 fault during a stage 1 translation, and the part of the ISS field that might contain details of the instruction is invalid. For more information, see Use of the HSR. Note On most systems, such a mapping to a Device memory type on the stage 2 translation is likely to indicate a Guest RTOS error, where the stage 1 translation is corrupted. Therefore, it is appropriate to trap this access to the hypervisor.
I _{JRZBY}	A cache maintenance instruction executed at PL1 can cause a stage 1 translation that might generate a stage 2 Permission fault, as described in this section. However: • If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a cache maintenance by VA instruction can generate a Permission fault in this way. • If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether a data or unified cache clean by VA to the Point of Unification instruction can generate a Permission fault in this way. Note This is an exception to the general rule that a cache maintenance instruction cannot generate a Permission fault.

D5.5.3 Alignment faults

R _{PHJYP}	The Arm memory architecture requires support for strict alignment checking.
I _{XGSQB}	This checking is controlled by: • SCTLR.A, for accesses made from any PE mode other than Hyp mode. • HSCTLR.A, for accesses made from Hyp mode. In addition, some instructions do not support unaligned accesses, regardless of the value of SCTLR.A or HSCTLR.A.
I _{FDDCX}	Unaligned data access defines when Alignment faults are generated, for both values of SCTLR.A or HSCTLR.A .
R _{VZBNX}	An Alignment fault can occur on an access for which the stage of address translation is disabled.
R _{DSWST}	Any unaligned access to memory region with any Device memory type attribute generates an Alignment fault.
I _{HLKPM}	Routing of aborts taken to AArch32 state defines the mode to which an Alignment fault is taken. The prioritization of Alignment faults depends on whether the fault was generated because of an access to a Device memory type, or for another reason. For more information, see AArch32 state prioritization of synchronous aborts from a single stage of address translation.

D5.5.4 External abort on a translation

R _{XWGCJ}	An External abort on a translation can be either synchronous or asynchronous. For more information on External aborts, see External aborts.
R _{WBCCQ}	An External abort on a translation is reported: • If the External abort is synchronous, using: — A synchronous Prefetch Abort exception if the translation is for an instruction fetch.

- A synchronous Data Abort exception if the translation is for a data access.
- If the External abort is asynchronous, using an SError exception, which is taken as an asynchronous Data Abort exception.

If an implementation reports the error in the translation asynchronously from executing the instruction whose instruction fetch or memory access caused the translation, these aborts behave essentially as interrupts. The aborts are masked when [PSTATE.A](#) is set to 1, otherwise they are reported using the Data Abort exception.

D5.5.5 AArch32 state prioritization of synchronous aborts from a single stage of address translation

IDDPZN	Exception prioritization for exceptions taken to AArch32 state describes the prioritization of exceptions taken from an Exception level that is using AArch32. This section gives additional information about the prioritization of MPU faults from PMSAv8-32 translation regimes.
RLBYFH	If a single instruction generates aborts on more than one memory access, the architecture does not define any prioritization between those aborts.
RKGQZN	In general, the Arm architecture does not define when asynchronous events are taken, and therefore the prioritization of asynchronous events is IMPLEMENTATION DEFINED.
IFNFZC	The priority numbering in this list only shows the relative priorities of aborts from a single stage of address translation in a PMSAv8-32 translation regime. This numbering has no global significance.
IFJVWP	For a single stage of translation in a PMSAv8-32 translation regime, the following numbered list shows the priority of the possible memory management faults on a memory access. In this list: • For memory accesses that undergo two stages of translation, the <i>italic entries show where the faults from the stage 2 translation can occur</i>. A stage 2 fault within a stage 1 translation follows the same prioritization of faults. • For synchronous External aborts from translations see also Synchronous External abort errors from address translation caching structures. The priority order, from highest priority to lowest priority, is: 1. Alignment fault not caused by memory type. This is possible for a stage 1 translation only. 2. Permission fault due to the input address being out of the address range to be translated or requiring an AArch32 Protection region that is disabled. 3. <i>Second stage abort on a stage 1 translation. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented.</i> This is second stage abort during a first stage translation. 4. Synchronous parity or ECC error. 5. Synchronous External abort. 6. Translation fault on a level 1 translation. 7. Alignment fault caused by the memory type. 8. Permission fault. 9. <i>A fault from the stage 2 translation of the memory access. When stage 2 address translation is enabled this includes an Address size fault caused by the PA being out of the range implemented.</i> 10. Synchronous parity or ECC error on the memory access. 11. Synchronous External abort on the memory access.

Note

The prioritization of IMPLEMENTATION DEFINED [MPU faults](#) for a Load-Exclusive or Store-Exclusive to an unsupported memory type is IMPLEMENTATION DEFINED.

See also [The MPU fault-checking sequence](#).

D5.6 Exception reporting in a PMSAv8-32 implementation

This section describes exception reporting, in AArch32 state, in a PMSAv8-32 implementation. That is, it describes only the reporting of exceptions that are taken to an Exception level that is using AArch32. EL2 provides an enhanced reporting mechanism for exceptions taken to the EL2 mode, Hyp mode. This means that, for PMSAv8-32, the exception reporting depends on the mode to which the exception is taken.

[About exception reporting](#) introduces the general approach to exception reporting, and the following sections then describe exception reporting at different privilege levels:

- [Reporting exceptions taken to PL1 modes.](#)
- [Fault reporting in PL1 modes.](#)
- [Summary of register updates on faults taken to PL1 modes.](#)
- [Reporting exceptions taken to Hyp mode.](#)
- [Use of the HSR.](#)
- [Summary of register updates on exceptions taken to Hyp mode.](#)

Note

The registers used for exception reporting also report information about debug exceptions. For more information, see:

- [Data Abort exceptions, taken to a PL1 mode.](#)
 - [Prefetch Abort exceptions, taken to a PL1 mode.](#)
 - [Reporting exceptions taken to Hyp mode.](#)
-

D5.6.1 About exception reporting

I_{YNLND}

Exceptions can be taken to:

- Hyp mode.
- A PL1 mode.

Exception reporting in Hyp mode differs significantly from that in the other modes, but in general, exception reporting returns:

- Information about the exception:
 - On taking an exception to Hyp mode, the *Hyp Syndrome Register*, [HSR](#), returns syndrome information.
 - On taking an exception to any other mode, a *Fault Status Register* (FSR) returns status information.
- For synchronous exceptions, one or more addresses associated with the exceptions, returned in *Fault Address Registers* (FARs). For a permitted exception to this requirement see [Fault address reporting on synchronous External aborts](#).

In all modes, additional IMPLEMENTATION DEFINED registers can provide additional information about exceptions.

Note

- [PE mode for taking exceptions](#) describes how the mode to which an exception is taken is determined.
- EL2 provides:
 - Specific exception types, that can only be taken from PL1 and EL0 modes, and are always taken to Hyp mode.
 - Routing controls that can route some exceptions from PL1 and EL0 modes to Hyp mode.

These exceptions are reported using the same mechanism as the Hyp mode reporting of PMSAv8-32 memory aborts, as described in this section.

I_{LQFTF}

Memory system faults generate either a Data Abort exception or a Prefetch Abort exception, as summarized in:

- [Reporting exceptions taken to PL1 modes.](#)
- [Memory fault reporting in Hyp mode.](#)

On an access that might have multiple aborts, the [MPU fault](#) checking sequence and the prioritization of aborts determine which abort occurs. For more information, see [The MPU fault-checking sequence](#) and [AArch32 state prioritization of synchronous aborts from a single stage of address translation](#).

D5.6.1.1 Fault address reporting on synchronous External aborts

R_{TWYFP}

The general architectural requirement is that, on a synchronous abort, the faulting address is recorded in a *Fault Address Register* (FAR). This requirement is relaxed for the case of a synchronous External abort that is not a synchronous External abort on a translation. In this case only:

- It is IMPLEMENTATION DEFINED whether the faulting address is recorded in a FAR.
- A bit in a fault reporting register, the FnV bit, indicates whether a valid address is recorded.

The details of this reporting depend on whether the exception is taken to:

- A PL1 mode, as described in [Reporting exceptions taken to PL1 modes](#).
- Hyp mode, as described in [Reporting exceptions taken to Hyp mode](#).

D5.6.2 Reporting exceptions taken to PL1 modes

I_{WBXJS}

The following sections give general information about the reporting of exceptions when they are taken to a PL1 mode:

- [Registers used for reporting exceptions taken to PL1 modes](#).
- [Data Abort exceptions, taken to a PL1 mode](#).
- [Prefetch Abort exceptions, taken to a PL1 mode](#).

[Fault reporting in PL1 modes](#) then describes the fault reporting in these modes, including the encodings used for reporting the faults.

Note

Security state, Exception levels, and AArch32 execution privilege describes how the PL1 modes map onto the Exception levels.

D5.6.2.1 Registers used for reporting exceptions taken to PL1 modes

I_{PQQHX}

AArch32 state defines the following registers, and register encodings, for exceptions taken to PL1 modes:

- The [DFSR](#) holds information about a Data Abort exception.
- The [DFAR](#) holds the faulting address for some synchronous Data Abort exceptions.
- The [IFSR](#) holds information about a Prefetch Abort exception.
- The [IFAR](#) holds the faulting address for some synchronous Prefetch Abort exceptions.

In addition, if implemented, the optional ADFSR and AIFSR can provide additional fault information, see [Auxiliary Fault Status Registers](#).

D5.6.2.1.1 Auxiliary Fault Status Registers

I_{VGHVB}

AArch32 state defines the following Auxiliary Fault Status Registers:

- The Auxiliary Data Fault Status Register, [ADFSR](#).
- The Auxiliary Instruction Fault Status Register, [AIFSR](#).

The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED. An implementation can use these registers to return additional fault status information. An example use of these registers is to return more information for diagnosing parity or ECC errors.

An implementation that does not need to report additional fault information must implement these registers as RES0. This ensures that an attempt to access these registers from software executing at PL1 does not cause an Undefined Instruction exception.

D5.6.2.2 Data Abort exceptions, taken to a PL1 mode

R_{KYSNC}

On taking a Data Abort exception to a PL1 mode:

- The [DFSR](#) is updated with details of the fault, including the appropriate Fault status code. If the Data Abort exception is synchronous, [DFSR.WnR](#) is updated to indicate whether the faulted instruction was a read or a write. However, if the fault is on a cache maintenance instruction, or on an address translation instruction, WnR is set to 1, to indicate a fault on a write instruction, and the CM bit is set to 1. If the Data Abort is external, then [DFSR](#) provides fields for additional classification of the abort, see [Provision for classification of External aborts](#). See the register description for more information about the returned fault information. See also [Data Abort on a Watchpoint exception](#). If the Data Abort exception is

- Synchronous, the [DFAR](#) is updated with the VA that caused the exception, but see [Fault address reporting on synchronous External aborts](#) for a permitted exception to this requirement.
- Asynchronous, the [DFAR](#) becomes UNKNOWN.

[DFSR.WnR](#) and [DFSR.CM](#) are UNKNOWN on an asynchronous Data Abort exception.

D5.6.2.2.1 Data Abort on an instruction cache or branch predictor maintenance instruction by VA

R_{CKYGN}

If an instruction cache invalidation by VA or branch predictor invalidation by VA operation generates a Data Abort exception that is taken to a PL1 mode, the [DFAR](#) is updated to hold the faulting VA.

The faulting is reported as follows:

- [DFSR.CM](#) is set to 1, to indicate a fault on a cache maintenance instruction.
- [DFSR.STATUS](#) indicates the cause of the fault, a Translation or Access flag fault.
- [DFSR.WnR](#) is set to 1.
- The [IFSR](#) is UNKNOWN.

D5.6.2.2.2 Data Abort on a Watchpoint exception

R_{HVTNX}

On taking a Data Abort exception caused by a watchpoint:

- [DFSR.FS](#) is updated to indicate a debug exception.
- [DFSR](#).{WnR, Domain} are UNKNOWN.
- [DFAR](#) is set to the address that generated the watchpoint.

Note

- [lr_ABT](#) indicates the address of the instruction that triggered the watchpoint.
-

R_{WZTWL}

A watchpointed address can be any byte-aligned address. The address reported in [DFAR](#) might not be the watchpointed address, and:

- For a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:
 - The lowest address accessed by the instruction that triggered the watchpoint.
 - The highest watchpointed address accessed by that instruction.

If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated. The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

Note

In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.

- For a watchpoint due to a Data Cache operation, the address is the address passed to the instruction. This might be an address that is above the watchpointed location.

D5.6.2.3 Prefetch Abort exceptions, taken to a PL1 mode

R_{WDQKK}

For a Prefetch Abort exception generated by an instruction fetch, the Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction, no Prefetch Abort exception is generated.

I_{NBBBM}

In addition, Breakpoint Instruction, Breakpoint, and Vector Catch exceptions, generate a Prefetch Abort exception, see the following for more information:

- [Exception syndrome information and preferred return address for a BKPT instruction.](#)
- [Exception syndrome information and preferred return address for a Breakpoint exception.](#)
- [Exception syndrome information and preferred return address for a Vector Catch exception.](#)

Note

Usually, the term *exception syndrome* is used only for exceptions taken to Hyp mode. The referenced sections use the term more generally, to include exception information reported in the [IFSR](#).

R_{PCNKL}

On taking a Prefetch Abort exception to a PL1 mode:

- The [IFSR](#) is updated with details of the fault, including the appropriate fault code. If appropriate, the fault code indicates that the exception was generated by a debug exception. See the register description for more information about the returned fault information.
- For a Prefetch Abort exception generated by an instruction fetch, the [IFAR](#) is updated with the VA that caused the exception, but see [Fault address reporting on synchronous External aborts](#) for a permitted exception to this requirement.
- For a Prefetch Abort exception generated by a debug exception, the [IFAR](#) is UNKNOWN.

D5.6.3 Fault reporting in PL1 modes

I_{GHVTT}

The FSRs provide fault information, including an indication of the fault that occurred. The following subsection describes fault reporting in PL1 modes for [PL1 fault reporting](#).

Reserved encoding in the IFSR and DFSR encodings tables gives some additional information about the encodings for both formats.

[Summary of register updates on faults taken to PL1 modes](#) shows which registers are updated on each of the reported faults.

D5.6.3.1 PL1 fault reporting

I_{LQYSC}

This subsection describes the fault reporting for a fault taken to a PL1 mode.

R_{CMTTH}

When the PE takes an exception, bit[9] of the FSR is set to 1 if the PE is using this FSR format.

The FSRs encode the fault in a 6-bit STATUS field, that comprises FSR[5:0]. In addition:

- For a fault taken to a PL1 mode, [Summary of register updates on faults taken to PL1 modes](#) shows whether the corresponding FAR is updated on the fault. That is:
 - For a fault reported in the [IFSR](#), whether the [IFAR](#) holds a valid address.
 - For a fault reported in the [DFSR](#), whether the [DFAR](#) holds a valid address.
- For a fault taken to the Hyp mode, [Summary of register updates on exceptions taken to Hyp mode](#) shows what registers are updated on the fault.

R_{ZNVVV}

The fault encodings reserve a single encoding for Cache faults. The details of these faults and any associated subsidiary registers are IMPLEMENTATION DEFINED.

D5.6.4 Summary of register updates on faults taken to PL1 modes

R_{JZVBX}

For faults that generate exceptions that are taken to a PL1 mode, [Table D5-8](#) shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

Table D5-8 Effect of a fault taken to a PL1 mode on the reporting registers

Fault	IFSR	IFAR	DFSR	DFAR
Faults reported as Prefetch Abort exceptions:				

Fault	IFSR	IFAR	DFSR	DFAR
MPU fault , always synchronous	Yes	Yes	-	-
Synchronous External abort on translation	Yes	Yes	-	-
Synchronous parity or ECC error on translation	Yes	Yes	-	-
Synchronous External abort	Yes	IMP DEF ^a	-	-
Synchronous parity or ECC error on memory access	Yes	Yes	-	-
Fault reported as Data Abort exception:				
Alignment fault, always synchronous	-	-	Yes	Yes
MPU fault , always synchronous	-	-	Yes	Yes
Fault on instruction cache maintenance	UNK	-	Yes	Yes
Synchronous External abort	-	-	Yes	IMP DEF ^a
Synchronous parity or ECC error on memory access	-	-	Yes	Yes
SError exception	-	-	Yes	UNK
SError exception from a parity or ECC error on memory access	-	-	Yes	UNK
Debug exceptions:				
Breakpoint, Breakpoint Instruction, or Vector Catch ^b	Yes	UNK	-	-
Watchpoint ^c	-	-	Yes	Yes

a. IMPLEMENTATION DEFINED. The [IFSR.FnV](#) or [DFSR.FnV](#) bit indicates whether the register holds a valid address. See [Fault address reporting on synchronous External aborts](#).

b. Generates a Prefetch Abort exception.

c. Generates a Data Abort exception.

D5.6.5 Reporting exceptions taken to Hyp mode

I_{ZNTNS}

Hyp mode is the EL2 mode. It is entered by taking an exception to Hyp mode.

The following exceptions are taken to Hyp mode:

- SError exceptions, IRQ exceptions, and FIQ exceptions, from PL1 and EL0 modes, and if configured by the AMO, FMO or IMO bits. For more information, see [Asynchronous exception routing controls](#).
 - When [HCR.TGE](#) is set to 1, all exceptions that would be routed to PL1 modes. For more information, see [Routing exceptions from EL0 to EL2](#).
 - When [HDCR.TDE](#) is set to 1, any debug exception that would otherwise be taken to a PL1 mode, see [Routing debug exceptions to EL2 using AArch32](#).
 - The privilege rules for taking exceptions mean that any exception taken from Hyp mode, must be taken to Hyp mode.
 - An abort that [Routing of aborts taken to AArch32 state](#) identifies as taken to Hyp mode.
 - Hypervisor Call exceptions, and Hyp Trap exceptions, are always taken to Hyp mode. These exceptions are supported only as part of EL2.
- Various operations from PL1 and EL0 modes can be *trapped* to Hyp mode, using the Hyp Trap exception. For more information, see [EL2 configurable controls](#).

R_{HDC}CLH

Synchronous exceptions taken to Hyp mode provide syndrome information in the [HSR](#).

R_QRSWR

On an abort exception taken to Hyp mode, the syndrome information in the [HSR](#) includes the Fault status code otherwise provided by the fault status register, and extends the fault reporting compared to that available for an exception taken to a PL1 mode.

In addition, for a Debug exception taken to Hyp mode, [DBGDSCRint.MOE](#) or [DBGDSCRext.MOE](#) shows what caused the Debug exception. This field is valid regardless of whether the Debug exception was taken from Hyp mode or from another mode.

For more information, see the following subsections:

- [Registers used for reporting exceptions taken to Hyp mode.](#)
- [Memory fault reporting in Hyp mode.](#)
- [Use of the HSR](#)

D5.6.5.1 Registers used for reporting exceptions taken to Hyp mode

ICXTVC

The following registers are used for reporting exceptions taken to Hyp mode:

- The [HSR](#) holds syndrome information for the exception.
- The [HDFAR](#) holds the VA associated with a Data Abort exception.
- The [HIFAR](#) holds the VA associated with a Prefetch Abort exception.
- The [HPFAR](#) holds bits[39:12] of the IPA associated with some aborts on stage 2 address translations.

In addition, if implemented, the optional [HADFSR](#) and [HAIFSR](#) can provide additional fault information, see [Hyp Auxiliary Fault Syndrome Registers](#).

D5.6.5.1.1 Hyp Auxiliary Fault Syndrome Registers

IGYJPF

EL2 also defines encodings for the following Hyp Auxiliary Fault Syndrome Registers:

- The Hyp Auxiliary Data Fault Syndrome Register, [HADFSR](#).
- The Hyp Auxiliary Instruction Fault Syndrome Register, [HAIFSR](#).

An implementation can use these registers to return additional fault status information for aborts taken to Hyp mode. They are the Hyp mode equivalents of the registers described in [Auxiliary Fault Status Registers](#). An example use of these registers is to return more information for diagnosing parity or ECC errors.

ICJQZJ

The architectural requirements for the [HADFSR](#) and [HAIFSR](#) are:

- The position of these registers is architecturally-defined, but the content and use of the registers is IMPLEMENTATION DEFINED.
- An implementation with no requirement for additional fault reporting can implement these registers as RES0, but the architecture does not require it to do so.

D5.6.5.2 Memory fault reporting in Hyp mode

RWMVVZ

Prefetch Abort and Data Abort exceptions taken to Hyp mode report memory faults.

ILRNYS

For these aborts, the [HSR](#) contains the following fault status information:

- The [HSR](#).EC field indicates the type of abort, as [Table D5-9](#) shows.
- The [HSR](#).ISS field holds more information about the abort. In particular:
 - Bits[5:0] of this field hold the STATUS field for the abort, using the encodings defined in [PL1 fault reporting with the Long-descriptor translation table format](#).
 - Other subfields of the ISS give more information about the exception, equivalent to the information returned in the FSR for a memory fault reported at PL1.

See the descriptions of the ISS fields for the memory faults, referenced from the *Syndrome description* column of [Table D5-9](#), for information about the returned fault information.

Table D5-9 HSR.EC encodings for aborts taken to Hyp mode

HSR.EC	Abort	Syndrome description
0x20	Prefetch Abort taken from PL1 or EL0 mode	ISS encoding for exception from a Prefetch Abort
0x21	Prefetch Abort taken from Hyp mode	
0x24	Data Abort taken from PL1 or EL0 mode	ISS encoding for exception from a Data Abort
0x25	Data Abort taken from Hyp mode	

For more information, see [Use of the HSR](#).

RZVFML

A Prefetch Abort exception is taken synchronously with the instruction that the abort is reported on. This means:

- If the PE attempts to execute the instruction a Prefetch Abort exception is generated.
- If an instruction fetch is issued but the PE does not attempt to execute the prefetched instruction, no Prefetch Abort exception is generated for that instruction. For example, if the execution flow branches round a prefetched instruction that would abort if the PE attempted to execute it, no Prefetch Abort exception is generated.

D5.6.5.2.1 Register updates on exception reporting in Hyp mode

The use of the [HSR](#), and of the other registers listed in [Registers used for reporting exceptions taken to Hyp mode](#), depends on the cause of the Abort. In reporting these faults, in general:

- If the fault generates a synchronous Data Abort exception, the [HDFAR](#) holds the associated VA, but see [Fault address reporting on synchronous External aborts](#) for a permitted exception to this requirement.
- If the fault generates a Prefetch Abort exception, the [HIFAR](#) holds the associated VA, but see [Fault address reporting on synchronous External aborts](#) for a permitted exception to this requirement.
- In the following cases, the [HPFAR](#) holds the faulting IPA:
 - A Translation or Access flag fault on a stage 2 translation.
 - A Translation, Access flag, or Permission fault on the stage 2 translation of an address accessed in a stage 1 translation.
 - A stage 2 Address size fault.
 In all other cases, the [HPFAR](#) is UNKNOWN.
- On a Data Abort exception that is taken to Hyp mode, the [HIFAR](#) is UNKNOWN.
- On a Prefetch Abort exception that is taken to Hyp mode, the [HDFAR](#) is UNKNOWN.

In addition, the reporting of particular aborts is as follows:

Abort on the stage 1 translation for a memory access from Hyp mode

The [HDFAR](#) or [HIFAR](#) holds the VA that caused the fault. The STATUS subfield of [HSR](#).ISS indicates the type of fault, Translation, Address size, Access flag, or Permission. The [HPFAR](#) is UNKNOWN.

Abort on the stage 2 translation for a memory access from a PL1 or EL0 mode

This includes aborts on the stage 2 translation of a memory access made as part of a translation for a stage 1 translation. The [HDFAR](#) or [HIFAR](#) holds the VA that caused the fault. The STATUS subfield of [HSR](#).ISS indicates the type of fault, Translation, Address size, Access flag, or Permission.

For any Access flag fault or Translation fault, and also for any Permission fault on the stage 2 translation of a memory access made as part of a translation for a stage 1 translation, the [HPFAR](#) holds the IPA that caused the fault. Otherwise, the [HPFAR](#) is UNKNOWN.

Abort caused by a synchronous External abort, or synchronous parity or ECC error, and taken to Hyp mode

The [HDFAR](#) or [HIFAR](#) holds the VA that caused the fault, but see [Fault address reporting on synchronous External aborts](#) for a permitted exception to this requirement. The [HPFAR](#) is UNKNOWN.

Data Abort caused by a Watchpoint exception and routed to Hyp mode because HDCR.TDE is set to 1

When [HDCR](#).TDE is set to 1, a Watchpoint exception generated in a PL1 or EL0 mode, that would otherwise generate a Data Abort exception, is routed to Hyp mode and generates a Hyp Trap exception. [HDFAR](#) is set to the address that generated the watchpoint.

Note

[ELR_hyp](#) indicates the address of the instruction that triggered the watchpoint.

A watchpointed address can be any byte-aligned address. The address reported in [HDFAR](#) might not be the watchpointed address, and, for a watchpoint due to an operation other than a Data Cache maintenance instruction, can be any address between and including:

- The lowest address accessed by the instruction that triggered the watchpoint.
- The highest watchpointed address accessed by that instruction.

If multiple watchpoints are set in this range, there is no guarantee of which watchpoint is generated.

Note

In particular, there is no guarantee of generating the watchpoint with the lowest address in the range.

The address must also be within a naturally-aligned block of memory of an IMPLEMENTATION DEFINED power-of-two size, containing a watchpoint address accessed by that location.

See also [Watchpoint exceptions](#).

In all cases, [HPFAR](#) is UNKNOWN.

Prefetch Abort caused by a Breakpoint Instruction exception and taken to Hyp mode

This abort is generated if a BKPT instruction is executed in Hyp mode. The abort leaves the HIFAR and HPFAR UNKNOWN.

See also Breakpoint Instruction exceptions.

Prefetch Abort caused by a Breakpoint Instruction, Breakpoint, or Vector Catch exception, and routed to Hyp mode because HDCR.TDE is set to 1

When HDCR.TDE is set to 1, a debug exception, generated in a PL1 or EL0 mode, that would otherwise generate a Prefetch Abort exception, is routed to Hyp mode and generates a Hyp Trap exception.

The abort leaves the HIFAR and HPFAR UNKNOWN. This is identical to the reporting of a Prefetch Abort exception caused by a Debug exception on a BKPT instruction that is executed in Hyp mode.

Note

The difference between these two cases is:

- The Debug exception on a BKPT instruction executed in Hyp mode generates a Prefetch Abort exception, taken to Hyp mode, and reported in the HSR using EC value 0x21.
- Aborts generated because HDCR.TDE is set to 1 generate a Hyp Trap exception, and are reported in the HSR using EC value 0x20.

D5.6.5.3 Use of the HSR

I_{RY}SP

The HSR holds syndrome information for any synchronous exception taken to Hyp mode. Compared with the reporting of exceptions taken to PL1 modes, the HSR:

- Always provides details of the fault. The DFSR and IFSR are not used.
- Provides more extensive information, for a wider range of exceptions.

Note

IRQ and FIQ exceptions taken to Hyp mode do not report any syndrome information in the HSR.

I_{JX}VYP

The HSR comprises:

- A 6-bit Exception class field, EC, that indicates the cause of the exception.
- An instruction length bit, IL. When an exception is caused by trapping an instruction to Hyp mode, this bit indicates the length of the trapped instruction, as follows:
 - 0 16-bit instruction trapped.
 - 1 32-bit instruction trapped.

In other cases the IL field is not valid and is RES1.
- An instruction specific syndrome field, ISS. Architecturally, this field could be defined independently for each defined Exception class (EC), but in practice several ISS formats are common to more than one EC.

The format of the HSR depends on the value of the EC field, as follows:

$0b000000 < EC \leq 0b001100$

The ISS part of the returned value includes the CV and COND fields described in Encoding of ISS[24:20] when $0b000000 < EC \leq 0b001100$. Figure D5-4 shows the HSR format in this case.

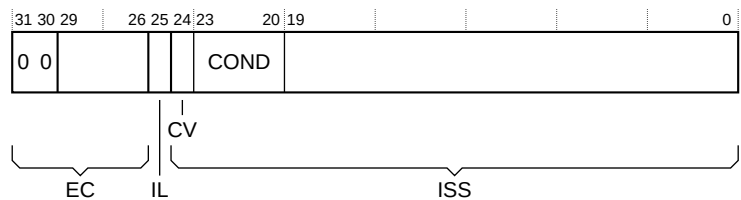


Figure D5-4 HSR format when the ISS includes CV and COND fields

$EC = 0b000000$ or $EC = 0b001110$

There are no generic fields within the ISS. Figure D5-5 shows the HSR format in this case.

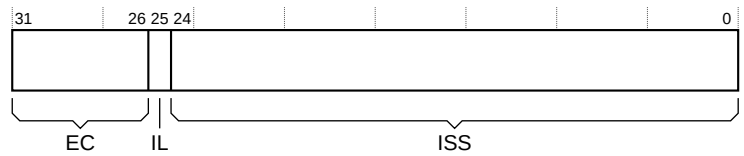


Figure D5-5 HSR format when the ISS does not include a COND field

D5.6.5.3.1 Encoding of ISS[24:20] when $0b000000 < EC \leq 0b001100$

I_{LRWPS}

For EC values that are nonzero and less than or equal to $0b001100$, ISS[24:20] provides the Condition code field for the trapped instruction, together with a valid flag for this field. The encoding of this part of the ISS field is:

CV, ISS[24] Condition code valid. Possible values of this bit are:

- 0** The COND field is not valid.
- 1** The COND field is valid.

COND, ISS[23:20]

The Condition code for the trapped instruction. This field is valid only when CV is set to 1.

If CV is set to 0, this field is RES0.

The full descriptions of the [HSR.ISS](#) formats give more information about the CV field.

———— Note ————

In some circumstances, it is IMPLEMENTATION DEFINED whether a conditional instruction that fails its Condition code check generates an Undefined Instruction exception, see [Conditional execution of undefined instructions](#).

D5.6.5.3.2 HSR exception classes

I_{YXQCP}

[Table D5-10](#) shows the encoding of the [HSR](#) exception class field, EC. Values of EC not shown in the table are reserved. For each EC value, the table references a subsection of the description of the [HSR](#) that describes the associated ISS format and gives information about the cause of the exception, for example the configuration required to enable the associated trap.

Table D5-10 HSR.EC field encoding

EC	Exception class	ISS description, or notes
0b000000	Unknown reason	ISS encoding for exceptions with an unknown reason.
0b000001	Trapped WFI or WFE instruction	ISS encoding for exception from a WFI or WFE instruction.
0b000011	Trapped MCR or MRC access with (coproc == 0b1111)	ISS encoding for exception from an MCR or MRC access.
0b000100	Trapped MCRR or MRRC access with (coproc == 0b1111)	ISS encoding for exception from an MCRR or MRRC access.
0b000101	Trapped MCR or MRC access with (coproc == 0b1110)	ISS encoding for exception from an MCR or MRC access.
0b000110	Trapped LDC or STC access	ISS encoding for exception from an LDC or STC instruction.
0b000111	Advanced SIMD or floating-point functionality trapped by a HCPTR . {TASE, TCP10} control	ISS encoding for exception from an access to SIMD or floating-point functionality, resulting from HCPTR.
0b001000	Trapped VMRS access, from ID group traps, that is not reported using EC 0b000111	ISS encoding for exception from an MCR or MRC access. This trap is not taken if the HCPTR settings trap the access.
0b001100	Trapped MRRC access with (coproc == 0b1110)	ISS encoding for exception from an MCRR or MRRC access.
0b001110	Illegal exception return to AArch32 state	ISS encoding for exception from an Illegal state or PC alignment fault.

EC	Exception class	ISS description, or notes
0b010001	Exception on SVC execution in AArch32 state routed to EL2	ISS encoding for exception from HVC or SVC instruction execution.
0b010010	HVC instruction execution in AArch32 state, when HVC is not disabled	
0b010011	Trapped execution of SMC instruction in AArch32 state	ISS encoding for exception from SMC instruction execution.
0b100000	Prefetch Abort from a lower Exception level	ISS encoding for exception from a Prefetch Abort.
0b100001	Prefetch Abort taken without a change in Exception level	
0b100010	PC alignment exception.	ISS encoding for exception from an Illegal state or PC alignment fault.
0b100100	Data Abort from a lower Exception level	ISS encoding for exception from a Data Abort.
0b100101	Data Abort taken without a change in Exception level	

All EC encodings not shown in [Table D5-9](#) are reserved by Arm.

D5.6.6 Summary of register updates on exceptions taken to Hyp mode

IC_{YMTK}

For memory system faults that generate exceptions that are taken to Hyp mode, [Table D5-11](#) shows the registers affected by each fault. In this table:

- Yes indicates that the register is updated.
- UNK indicates that the fault makes the register value UNKNOWN.
- A null entry, -, indicates that the fault does not affect the register.

Note

For a list of the [MPU faults](#), see [Types of MPU faults](#).

Table D5-11 Effect of an exception taken to Hyp mode on the reporting registers

Fault	HSR	HIFAR	HDFAR	HPFAR
Faults reported as Prefetch Abort exceptions:				
MPU fault ^a at stage 1.	Yes	Yes	UNK	UNK
Translation or Access flag MPU fault ^a at stage 2.	Yes	Yes	UNK	Yes
Other ^b MPU fault ^a at stage 2.	Yes	Yes	UNK	UNK
Stage 2 MPU fault ^a on a stage 1 translation.	Yes	Yes	UNK	Yes
Synchronous External abort on translation.	Yes	Yes	UNK	UNK
Synchronous parity or ECC error on translation.	Yes	Yes	UNK	UNK
Synchronous External abort.	Yes	IMP DEF ^c	UNK	UNK
Synchronous parity or ECC error on memory access.	Yes	Yes	UNK	UNK
Fault reported as Data Abort exception:				
MPU fault ^a at stage 1.	Yes	UNK	Yes	UNK
Translation or Access flag MPU fault ^a at stage 2.	Yes	UNK	Yes	Yes
Other ^b MPU fault ^a at stage 2.	Yes	UNK	Yes	UNK
Stage 2 MPU fault ^a on a stage 1 translation.	Yes	UNK	Yes	Yes

Fault	HSR	HIFAR	HDFAR	HPFAR
Synchronous External abort on translation.	Yes	UNK	Yes	UNK
Synchronous parity or ECC error on translation.	Yes	UNK	Yes	UNK
Synchronous External abort.	Yes	UNK	IMP DEF ^c	UNK
Synchronous parity or ECC error on memory access.	Yes	UNK	Yes	UNK
SError exception.	Yes	UNK	UNK	UNK
SError exception from a parity or ECC error on memory access.	Yes	UNK	UNK	UNK
Debug exception:				
Breakpoint Instruction ^d , generates a Prefetch Abort exception.	Yes	UNK	-	UNK
Debug exception routed to Hyp mode because HDCR.TDE is set to 1. Generates a Hyp Trap exception.				
Breakpoint Instruction or Vector Catch.	Yes	UNK	-	UNK
Watchpoint.	Yes	-	Yes	UNK

- a. For more information, see [Classification of MPU faults taken to Hyp mode](#)
b. [MPU fault](#) other than a Translation fault or an Access flag fault.
c. IMPLEMENTATION DEFINED. The FnV bit in the HSR.ISS field indicates whether the register holds a valid address. See [Fault address reporting on synchronous External aborts](#).
d. All other debug exceptions are not permitted in Hyp mode.

Note

Unlike [Table D5-8](#), the Hyp mode fault reporting table does not include an entry for a fault on an instruction cache maintenance instruction. That is because, when the fault is taken to Hyp mode, the reporting indicates the cause of the fault, for example a Translation fault, and ISS.CM is set to 1 to indicate that the fault was on a cache maintenance instruction, see [ISS encoding for exception from a Data Abort](#).

D5.6.6.1 Classification of MPU faults taken to Hyp mode

This subsection gives more information about the [MPU faults](#) shown in [Table D5-11](#).

Note

All [MPU faults](#) are synchronous.

IVBTXM

The table uses the following descriptions for [MPU faults](#) taken to Hyp mode:

MPU fault at stage 1

This is an [MPU fault](#) generated on a stage 1 translation performed in the EL2 translation regime.

MPU fault at stage 2

This is an [MPU fault](#) generated on a stage 2 translation performed in the PL1&0 translation regime.

As the table shows, for the faults in this group:

- Translation and Access flag faults update the [HPFAR](#).
- Permission faults leave the [HPFAR](#) UNKNOWN.

MPU stage 2 fault on a stage 1 translation

This is an [MPU fault](#) generated on the stage 2 translation of an address accessed in a stage 1 translation performed in the PL1&0 translation regime. For more information about these faults see [Stage 2 fault on a stage 1 translation](#).

[Figure D5-1](#) shows the different translation regimes and associated stages of translation.

D5.7 PMSAv8-32 implications for caches

RQXCyx

Enabling or reconfiguring the MPU, or reprogramming any protection regions, can result in new and different memory attributes for a previously accessed or speculatively accessed address. In this situation, the rules for [Mismatched memory attributes](#) apply.

D5.7.1 Cache line length

A PMSAv8-32 MPU permits the definition of protection regions that might be smaller than a cache line in the implementation. Therefore, the following rules apply:

- If the MPU is configured such that multiple differing attributes apply to a single cache line, then for any access to that cache line the rules for mismatched memory attributes apply. For more information, see Memory region attributes.
- Marking any part of a cache line as Write-Back permits the entire line to be treated as Write-Back.

Chapter D6

The Performance Monitors Extension

This chapter describes the implementation of the Arm Performance Monitors, which are an optional non-invasive debug component. It describes version 3 of the *Performance Monitor Unit* (PMU) architecture, [FEAT_PMUv3](#). It contains the following sections:

- [About the Performance Monitors](#)
- [Accuracy of the Performance Monitors](#)
- [Behavior on overflow](#)
- [Attributability](#)
- [Controlling the PMU counters](#)
- [Multithreaded implementations](#)
- [Event filtering](#)
- [Performance Monitors and Debug state](#)
- [Counter access](#)
- [Performance Monitors Extension registers](#)

D6.1 About the Performance Monitors

The Performance Monitors Extension is an OPTIONAL feature of an implementation, but Arm strongly recommends that implementations include version 3 of the Performance Monitors Extension, [FEAT_PMUv3](#).

Note

No previous versions of the Performance Monitors Extension can be implemented in architectures from Armv8.0.

The Performance Monitors Extension provides:

- The 64-bit cycle counter, [PMCCNTR](#). The cycle counter is always implemented.
- The event counters, [PMEVCNTR<n>](#). The number of event counters is an IMPLEMENTATION DEFINED parameter, [NUM_PMU_COUNTERS](#), between 0 and 31.
The event counters are 32 bit counters.
- Controls for:
 - Enabling counters. See [Enabling PMU counters](#).
 - Resetting counters. [Resetting counters](#).
 - Flagging overflows. See [Behavior on overflow](#).
 - Enabling interrupts on overflow. See [Generating overflow interrupt requests](#).
 - Prohibiting counting. See [Prohibiting counting](#).
 - Partitioning the implemented event counters into ranges. See [Event counter ranges](#).

The PMU architecture uses event numbers to identify an event. It:

- Defines event numbers for common events, for use across many architectures and microarchitectures. See [Common event numbers](#).

Note

Implementations that include [FEAT_PMUv3](#) must, as a minimum requirement, implement a subset of the common events. See [Common event numbers](#).

- Reserves a large event number space for IMPLEMENTATION DEFINED events. See [IMPLEMENTATION DEFINED event numbers](#).

The full set of events for an implementation is IMPLEMENTATION DEFINED. Arm recommends that implementations include all of the events that are appropriate to the architecture profile and microarchitecture of the implementation. For more information, see [PMU Event Descriptions](#).

When an implementation includes the Performance Monitors Extension, the architecture defines the following possible interfaces to the Performance Monitors Extension registers:

- A System register interface. This interface is mandatory.

Note

In AArch32 state, the interface is in the (`coproc == 0b1111`) encoding space.

- An external debug interface which optionally supports memory-mapped accesses. Implementation of this interface is OPTIONAL. See [Recommended External Interface to the Performance Monitors](#).

An operating system can use the System registers to access the counters.

Also, if required, the operating system can enable application software to access the counters. This enables an application to monitor its own performance with fine-grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

To enable interaction with external monitoring, an implementation might consider additional enhancements, such as providing:

- A set of events, from which a selection can be exported onto a bus for use as external events.
- The ability to count external events. This enhancement requires the implementation to include a set of external event input signals.

D6.1.1 Event counter ranges

I _{STBND}	If any of the following apply, then the behavior is CONSTRAINED UNPREDICTABLE: • HDCR.HPMN is 0. • HDCR.HPMN is greater-than NUM_PMU_COUNTERS. See: • The Performance Monitors Extension when accessing the Performance Monitors Extension in AArch32 state.
I _{SBQGI}	When EL2 is enabled in the current Security state, self-hosted software executing at EL1 and, if enabled by PMUSERENR, EL0 only has access through System registers to event counters in the first range, the cycle counter PMCCNTR. This software is not able to directly determine that event counters in the second range are implemented. Self-hosted software executing at EL2 additionally has access through System registers to event counters in the second range.
R _{NNBYQ}	If all of the following are true, then direct reads of PMCR.N return the Effective value of HDCR.HPMN: • PE is executing at EL0 or EL1. • The access does not generate an exception.

D6.1.2 Interaction with EL2

I _{FZHQW}	Software executing at EL2 can program HDCR.HPMN to partition the event counters into a first range and a second range. See Event counter ranges.
I _{BPVYL}	Software executing at EL2 can: • Trap an access at EL0 or EL1 to the PMU. This means the hypervisor can identify which Guest OSs are using the PMU and intelligently employ switching of the PMU state. There is a separate trap for the PMCR register. When EL2 is implemented, software executing at EL1 and, if enabled by PMUSERENR, EL0: • Will read the value of HDCR.HPMN for PMCR.N. • Cannot access the event counters in the second range, or the controls associated with them.
I _{MVYJY}	For more information, see: • Enabling PMU counters. • Counter access. • Controlling the PMU counters. • Multithreaded implementations.

D6.1.3 Interaction with trace

When [FEAT_ETMv4](#) is implemented, it is IMPLEMENTATION DEFINED whether the implementation exports PMU events to the ETM trace unit, and the form of any exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting is controlled by [PMCR.X](#).

For more information about when the trace unit is permitted to observe PMU events, see Controls to prohibit trace at Exception levels and External inputs.

D6.2 Accuracy of the Performance Monitors

The Performance Monitors:

- Are a non-invasive debug component. See [Non-invasive behavior](#).
- Must provide broadly accurate and statistically useful count information.

However, the Performance Monitors allow for:

- A reasonable degree of inaccuracy in the counts to keep the implementation and validation cost low. See [A reasonable degree of inaccuracy](#).
- IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, that software must configure before using certain PMU events. For example, to configure how the PE generates PMU events for components such as external caches and external memory.
- Other IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to optionally put the PE in an operating state that might do one or both of the following:
 - Change the level of non-invasiveness of the Performance Monitors so that enabling an event counter can impact the performance or behavior of the PE.
 - Allow inaccurate counts. This includes, but is not limited to, cycle counts.

D6.2.1 Non-invasive behavior

The Performance Monitors are a non-invasive debug feature. A non-invasive debug feature permits the observation of data and program flow. Performance Monitors, PC Sample-based Profiling and Trace are non-invasive debug features.

Non-invasive debug components do not guarantee that they do not make any changes to the behavior or performance of the processor. Any changes that do occur must not be severe however, as this will reduce the usefulness of event counters for performance measurement and profiling. This does not include any change to program behavior that results from the same program being instrumented to use the Performance Monitors, or from some other performance monitoring process being run concurrently with the process being profiled in a multitasking operating system. As such, a reasonable variation in performance is permissible.

Note

Power consumption is one measure of performance. Therefore, a reasonable variation in power consumption is permissible.

Arm does not define a *reasonable variation in performance*, but recommends that such a variation is kept within 5% of normal operating performance, when averaged across a suite of code that is representative of the application workload.

For some common architectural events, this requirement to be non-invasive can conflict with the requirement to present an accurate value of the count under normal operating conditions. Should an implementation require more performance-invasive techniques to accurately count an event, there are the following options:

- If the event is optional, define an alternative IMPLEMENTATION DEFINED event that accurately counts the event and document the impact on performance of enabling the event.
- Provide an IMPLEMENTATION DEFINED control that disables accurate counting of the event to restore broadly accurate performance, and document the impact on performance of accurate counting.

Note

The IMPLEMENTATION DEFINED invasiveness might include the PE disabling one or more performance optimizations such as multi-issue, out-of-order execution, and pipelining.

A software implication is that the performance invasiveness is proportional to the sampling interval. This can disproportionately affect profiling of rare events, because the population is smaller and therefore the interval will tend to be smaller.

D6.2.2 A reasonable degree of inaccuracy

The Performance Monitors provide broadly accurate and statistically useful count information. To keep the implementation, validation, and performance overhead costs low, a reasonable degree of inaccuracy in the counts and the detection of overflow is acceptable. Arm does not define a *reasonable degree of inaccuracy* but recommends the following guidelines:

- Under normal operating conditions, the counters must present an accurate value of the count.
- In exceptional circumstances, such as enabling or disabling the counter, entry to and exit from Debug state, a change in Security state, taking an exception, executing an exception return, or other boundary condition, it is acceptable for the count to be inaccurate, unless otherwise stated.
- Under very unusual, non-repeating pathological cases, the counts can be inaccurate. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is very unlikely.

Note

- An implementation must not introduce inaccuracies that can be triggered systematically by the execution of normal pieces of software. For example, it is not reasonable for the count of branch behavior to be inaccurate when caused by a systematic error generated by the loop structure producing a dropping in branch count. However, dropping a single branch count as the result of a rare interaction with an interrupt is acceptable.
 - To limit inaccuracy caused by entering Debug state, an implementation can, for example, promptly disable the counters during the Debug state entry sequence.
-

The architecture does not permit these behaviors for events that specifically count architectural exceptional circumstances, such as a change in Exception level, for example, [EXC_TAKEN](#).

The permitted inaccuracy limits the possible uses of the Performance Monitors. In particular, the architecture does not define the point in a pipeline where the event counter is incremented, relative to the point where a read of the event counters is made. This means that pipelining effects can cause some imprecision, and can affect which events are counted.

This applies to both architectural and microarchitectural events, and can apply differently to different events. Arm recommends that is applied consistently between counters. For example, any boundary condition effects that apply to an event counter counting the [CPU_CYCLES](#) event should apply in the same way to the cycle counter PMCCNTR_EL0. However, this is not required.

Inaccuracies around exceptional events, such as enabling or disabling the counter, should be such that the inaccuracy is reduced by lengthening the sampling period.

Where a direct write to a Performance Monitors control register disables a counter, and is followed by a [Context Synchronization event](#), any subsequent indirect read of the control register by the Performance Monitors to determine whether the counter is enabled will return the updated value. Any subsequent direct read of the counter or counter overflow status flags will return the value at the point the counter was disabled.

Note

The imprecision means that the counter might have counted an event around the time the counter was disabled, but does not allow the event to be observed as counted after the counter was disabled.

An implementation must document any particular scenarios where significant inaccuracies are expected.

D6.3 Behavior on overflow

IYVWXN	The event counters, PMEVCNTR<n> are 32-bit unsigned counters that overflow if incrementing PMEVCNTR<n> causes an unsigned overflow of an event counter, the PE sets PMOVSCLR_EL0[n] to 1. The cycle counter, PMCCNTR , is a 64-bit unsigned counter, if incrementing PMCCNTR causes an unsigned overflow of bits [31:0] of the cycle counter, the PE sets PMOVSCLR_EL0[31] to 1.
RHTPDY	The update of PMOVSCLR_EL0 occurs synchronously with the update of the counter.
IHRJFC	A PMU overflow can generate an interrupt request. For more information, see Generating overflow interrupt requests .
IPVBJQ	A PMU overflow can be used as an input trigger event to the <i>Cross Trigger Interface</i> (CTI) for the PE. For more information, see Performance Monitors overflow trigger event .

D6.3.1 Generating overflow interrupt requests

Software can program the Performance Monitors so that an overflow interrupt request is generated when a counter overflows. See [PMINTENSET](#) and [PMINTENCLR](#).

-
- Note**
- The mechanism by which an interrupt request from the Performance Monitors generates an FIQ or IRQ exception is IMPLEMENTATION DEFINED.
 - Arm recommends that the overflow interrupt requests:
 - Translate into a [PMUIRQ](#) signal, so that they are observable to external devices.
 - Connect to inputs on an IMPLEMENTATION DEFINED Generic Interrupt Controller as a *Private Peripheral Interrupt* (PPI) for the originating processor. See the *ARM Generic Interrupt Controller Architecture Specification* for information about PPIs.
 - Connect to a CTI, see [The Embedded Cross-Trigger Interface](#).
 - Arm strongly discourages implementations from connecting overflow interrupt requests from multiple PEs to the same *System Peripheral Interrupt* (SPI) identifier.
 - From GICv3, the *ARM® Generic Interrupt Controller Architecture Specification* recommends that the *Private Peripheral Interrupt* (PPI) with ID 23 is used for overflow interrupt requests.
-

Software can write to the counters to control the frequency at which interrupt requests occur. For example, software might set a 32-bit counter to `0xFFFF_0000`, to generate another counter overflow after 65 536 increments, and reset it to this value every time an overflow interrupt occurs.

-
- Note**
- If an event can occur multiple times in a single clock cycle, then counter overflow can occur without the counter registering a value of zero.
-

The overflow interrupt request is a level-sensitive request. The PE signals a request for:

- Any implemented [PMEVCNTR<n>](#) counter, when the [global enable control](#) for the event counter is 1, the value of [PMOVSSET.P<n>](#) is 1, and the value of [PMINTENSET.P<n>](#) is 1.
- The cycle counter, when the values of [PMCR.E](#), [PMOVSSET.C](#) and [PMINTENSET.C](#) are all 1.
- The instruction counter, when the values of [PMCR.E](#), [PMOVSSET_EL0.F0](#), and [PMINTENSET_EL1.F0](#), are all 1.

The overflow interrupt request is active in all Security states.

The interrupt handler for the counter overflow request must cancel the interrupt request, for example, by writing 1 to [PMOVSCLR_EL0.P<n>](#) to clear the overflow bit to 0.

D6.3.1.1 Pseudocode description of overflow interrupt requests

See [R-profile Architecture Pseudocode](#) for a pseudocode description of overflow interrupt requests. The [CheckForPMUOverflow\(\)](#) pseudocode function signal PMU overflow interrupt requests to an interrupt controller and PMU overflow trigger events to the cross-trigger interface.

D6.4 Attributability

An event caused by the PE counting the event is *Attributable*. If an agent other than the PE that is counting the events causes an event, these events are Unattributable.

An event is defined as being either Attributable or Unattributable. If the event is Attributable, it is further defined whether it is Attributable to:

- The current Exception level of the PE.
- When the PE is in Debug state, operations issued to the PE by the debugger through the external debug interface.

In a multithreaded implementation, an event might be generated by another PE with the same values for affinity level 1 and higher. This event is further defined as Attributable to:

- The current Exception level of that PE.
- When that PE is in Debug state, operations issued to that PE by the debugger through the external debug interface.

See [Multithreaded implementations](#) for information about enabling and restricting counting events in a multithreaded implementation.

Note

An implementation is described as multithreaded when the lowest level of affinity, as reported by [MPIDR](#) {Aff3, Aff2, Aff1, Aff0}, consists of logical PEs that are implemented using a multithreading type approach. In this section, when referring to a multithreaded implementation, *thread* is used to mean processing elements with:

- [MPIDR](#).MT set to 1,
 - Different values for affinity level 0.
 - The same values for affinity level 1 and higher.
-

An event can be defined as the combination of multiple subevents, which can be either Attributable or Unattributable.

All architecturally defined events are [Attributable](#), unless otherwise stated.

Unattributable events might be counted when Attributable events are not counted. For more information see, [Event filtering](#) and [Performance Monitors and Debug state](#).

D6.5 Controlling the PMU counters

This section describes the mechanisms available for controlling the PMU event and cycle counters. The following sections describe those mechanisms:

- [Enabling PMU counters.](#)
- [Prohibiting counting.](#)

D6.5.1 Enabling PMU counters

ICFQVP Counters are enabled by a combination of the *global enable control* for the counter and a counter enable control, [PMCNTENSET](#), as described in the following sections.

D6.5.1.1 Enabling the event counters

RKYBQW The global enable controls for PMU event counters are defined as follows:

- The [global enable control](#) for PMU event counters in the second range is [HDCR](#).HPME.
- The [global enable control](#) for PMU event counters in the first range is [PMCR](#).E.

ISTZYQ [PMCNTENSET](#).P<n> is the counter enable control for event counter *n*.

IVQXCB The [global enable control](#) is used, with other controls, for all of the following:

- Enabling counting of events by [PMEVCNTR](#)<n>.
- Generation of the PMU overflow interrupt request, [PMUIRQ](#), by [PMOVSLR_EL0](#)[n].
- Generation of the CTI overflow interrupt trigger by [PMOVSLR_EL0](#)[n].

The global enable control for the other range do not affect whether the event counter [PMEVCNTR](#)<n> generates any of these.

IQKDTZ The effect of [HDCR](#).{HPME, HPMN} on the counter enables applies at all Exception levels and in all Security states.

D6.5.1.2 Enabling the cycle counter

IBTGDI For the cycle counter, [PMCCNTR](#), the [global enable control](#) is [PMCR](#).E.
For the cycle counter, the counter enable control is [PMCNTENSET](#).C.
[PMCR](#).DP controls whether the cycle counter is disabled when event counting is prohibited, as described in [Prohibiting counting](#).

D6.5.2 Resetting counters

Software can reset all the event counters and/or the cycle counter in a single operation by writing 1 to [PMCR](#).C and/or [PMCR](#).P.

D6.5.3 Prohibiting counting

DJHDZC In this section:

- *Context* means the Exception level, and Debug state.
- Whether event counting for a context is allowed or prohibited by a PE counting an event is defined by the rules in this section.

[Attributability](#) defines how an event is [Attributable](#) to a context.

RQRRHL When all of the following apply:

- An event is to be counted by event counter <n>.
- The [Effective value](#) of [PMEVTYPER](#)<n>.MT is 0 and the event is [Attributable](#) to the PE counting the event.
- The PE counting the event is executing in a context **A**.
- The event is [Attributable](#) to context **B**.

Then, it is IMPLEMENTATION SPECIFIC which of context **A** or context **B** the PE counting the event uses to determine whether the event is allowed to be counted or is not counted.

Regardless of whether the event is allowed to be counted or not, if event filtering or any other controls disable counting the event, then the event is not counted.

It is IMPLEMENTATION SPECIFIC whether event filtering uses context **A** or context **B**.

R_{XBZL}

When all of the following apply:

- An event is to be counted by event counter <n>.
- The Effective value of `PMEVTYPEPER<n>.MT` is 0 and the event is [Attributable](#) to the PE counting the event.
- The PE counting the event is executing in a context **A**.
- The event is [Attributable](#) to context **B**.

Then all of the following apply:

- If both context **A** event counting and context **B** event counting are allowed by the PE counting the event, then the event is allowed to be counted.
- If context **A** event counting is prohibited by the PE counting the event and context **B** event counting is allowed by the PE counting the event, then it is IMPLEMENTATION SPECIFIC whether the event is allowed to be counted or not counted.
- If context **B** event counting is prohibited by the PE counting the event, then the event is not counted, regardless of whether context **A** event counting is allowed or prohibited by the PE counting the event.

Whether event counting for any context is prohibited by the PE the event is [Attributable](#) to has no effect on this rule.

Regardless of whether the event is allowed to be counted or not, if event filtering or any other controls disable counting the event, then the event is not counted. It is IMPLEMENTATION SPECIFIC which of context **A** or context **B** event filtering uses.

R_{RTMLK}

When all the following apply:

- An event is to be counted by event counter <n>.
- The Effective value of `PMEVTYPEPER<n>.MT` is 0 and the event is [Attributable](#) to a PE other than the PE counting the event.

Then the event is not counted. For more information, see [Multithreaded implementations](#).

I_{SLTTL}

[R_{QRRHL}](#) and [R_{XBZL}](#) allow events in a single-threaded implementation to be attributed to the current context when they are counted, rather than requiring the PE to keep track of precisely which context an event was generated in. This is acceptable because the current context changes rarely, meaning such an implementation exhibits [a reasonable degree of inaccuracy](#).

This would not be acceptable for multithreaded implementations where an event might be generated by a different PE from the PE counting the events. However, these rules permit an implementation to keep track of which PE generated the event and attribute the event to the current context of the PE counting the event. It is also not expected that the controls for prohibiting events in a given context will be set differently on different PEs in a multithreaded implementation that is making use of the PMU multithreading extensions.

R_{MMZLZ}

EL2 event counting is prohibited at EL2 unless any of the following is true:

- `FEAT_PMUV3p1` is not implemented.
- `HDCCR.HPMD` is 0.
- The event is being counted by an event counter <n> in the second range.

R_{TDGMF}

The accessibility of Performance Monitors registers is unaffected by whether event counting is enabled or prohibited.

R_{SGDTN}

The cycle counter, `PMCCNTR`, counts even when counting is prohibited, unless `PMCR.DP` is set to 1 or the PE is in Debug state.

R_{PTKLW}

Other counters can be enabled or disabled by using `PMCR.E` and `PMCNTENSET.C`.

I_{FZPQV}

The `CountPMUEvents(n)` function returns TRUE if any of the following apply:

- `PMEVCNTR<n>` counts events at the current Exception level and state.
- `n` is 31 and the cycle counter counts cycles at the current Exception level and state.
- Otherwise, the function returns FALSE.

However, this function does not completely describe the behavior for Unattributable events and events Attributable to a Security state that is not the current Security state, an Exception level that is not the current Exception level, or both. For more information, see [CountPMUEvents\(\)](#) in .

I_{BT}VLD

The Performance Monitors are intended to be broadly accurate and statistically useful, see [Accuracy of the Performance Monitors](#). Some inaccuracy is permitted at the point of changing between a state where counting is prohibited and a state where counting is allowed, however. To avoid the leaking of information, the permitted inaccuracy is that transactions that are not prohibited can be uncounted. Where possible, prohibited transactions must not be counted, but if they are counted, then that counting must not degrade security.

D6.6 Multithreaded implementations

If an implementation is multithreaded and the [Effective value](#) of `PMEVTYPER<n>.MT` is 1, then, unless otherwise stated, the counter counts the sum of all events across all PEs with the same level 1 Affinity. A pair of PEs have the same level 1 Affinity if they have the same values for all fields in `MPIDR_EL1` or `MPIDR` except the `Aff0` field.

See also [Cycle event counting in multithreaded implementations](#) for additional requirements for events that count cycles.

Events on other PEs are not counted when the [Effective value](#) of `PMEVTYPER<n>.MT` is 0.

If the CPU implements multithreading, it is IMPLEMENTATION DEFINED whether `PMEVTYPER<n>.MT` is implemented as `RW` or `RES0`.

See Time as measured by the Performance Monitors cycle counter.

D6.7 Event filtering

The PMU can filter events by various combinations of Exception level. This gives software the flexibility to count events across multiple processes.

D6.7.1 Filtering by Exception level

For an event that is [Attributable](#) to an Exception level, in a multithreaded implementation:

- When the [Effective value](#) of `PMEVTYPER<n>.MT` is 1, the specified filtering is evaluated using the current Exception level and Security state of the thread to which the event is [Attributable](#). See [Example D6-1](#).
- When the [Effective value](#) of `PMEVTYPER<n>.MT` is 0, the event is only counted if it is [Attributable](#) to the counting thread, and the filtering is evaluated using the Exception level and Security state of the counting thread.

Example D6-1

Example of the effect of the `PMEVTYPER<n>_EL0.MT` control

In a multithreaded implementation, if the [Effective value](#) of `PMEVTYPER<n>.MT` is 1 and the value of `PMEVTYPER<n>.U` is 1 on the counting thread, then event counter *n* does not count events [Attributable](#) to EL0 on another thread, even if the counting thread is not executing at EL0.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether the filtering applies. In a multithreaded implementation, if the filtering applies to an Unattributable event, then the filtering is evaluated using the Exception level and Security state of the counting thread.

In AArch32 state, the filtering controls are provided by the `PMEVTYPER<n>` and `PMCCFILTR` registers.

For more information, see the individual register descriptions and [Multithreaded implementations](#).

D6.7.2 Accuracy of event filtering

For most events, it is acceptable that, during a transition between states, events generated by instructions executed in one state are counted in the other state. The following sections describe the cases where event counts must not be counted in the wrong state:

- [Exception-related events](#).
- [Software increment events](#).

D6.7.2.1 Exception-related events

The PMU must filter events related to exceptions and exception handling according to the Exception level in which the event occurred. These events are:

- `EXC_TAKEN`, Exception taken.
- `EXC_RETURN`, Instruction architecturally executed, Condition code check pass, exception return.
- `CID_WRITE_RETIRED`, Instruction architecturally executed, Condition code check pass, write to `CONTEXTIDR`.
- `TTBR_WRITE_RETIRED`, Instruction architecturally executed, Condition code check pass, write to translation table base.
- `EXC_UNDEF`, Exception taken, other synchronous.
- `EXC_SVC`, Exception taken, Supervisor Call.
- `EXC_PABORT`, Exception taken, Instruction Abort.
- `EXC_DABORT`, Exception taken, Data Abort or SError.
- `EXC_IRQ`, Exception taken, IRQ.
- `EXC_FIQ`, Exception taken, FIQ.
- `EXC_SMC`, Exception taken, Secure Monitor Call.

- [EXC_HVC](#), Exception taken, Hypervisor Call.
- [EXC_TRAP_PABORT](#), Exception taken, Instruction Abort not [Taken locally](#).
- [EXC_TRAP_DABORT](#), Exception taken, Data Abort or SError not [Taken locally](#).
- [EXC_TRAP_OTHER](#), Exception taken, other traps not [Taken locally](#).
- [EXC_TRAP_IRQ](#), Exception taken, IRQ not [Taken locally](#).

The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions at EL0. Similarly, it is not acceptable for the PMU to count exception returns or writes to [CONTEXTIDR](#) after the return from the exception.

D6.7.2.2 Software increment events

The PMU must filter software increment events according to the Exception level in which the software increment occurred. Software increment counting must also be precise, meaning the PMU must count every architecturally executed software increment event, and must not count any [Speculatively executed](#) software increment.

If the PE performs two architecturally executed writes to the PMSWINC_EL0 or [PMSWINC](#) register without an intervening [Context Synchronization event](#), then the counter is incremented twice.

For more information, see [SW_INCR](#).

D6.7.3 Pseudocode description of event filtering

See [CountPMUEvents\(\)](#) in [R-profile Architecture Pseudocode](#) for a pseudocode description of event filtering. However, this function does not completely describe the behavior for Unattributable events.

D6.8 Performance Monitors and Debug state

Events that count cycles are not counted in Debug state.

Events [Attributable](#) to the operations issued by the debugger through the external debug interface are not counted in Debug state.

In an implementation that supports multithreading, when the [Effective value](#) of PMEVTYPER<n>_EL0.MT is 1, if an event is Attributable to an operation issued by the debugger through the external debug interface to another thread that is in Debug state, then the event is not counted, and it is IMPLEMENTATION DEFINED whether the event is counted when the counting thread is in Debug state.

For all other events, it is IMPLEMENTATION SPECIFIC whether it is counted when the counting PE is in Debug state. If the event might be counted, then the rules in [Filtering by Exception level](#) apply for the current Security state in Debug state.

D6.9 Counter access

- I_{VYCGK}** All implemented event counters are accessible in EL2. If EL2 is implemented the hypervisor uses [HDCR.HPMN](#) to reserve an event counter, with the effect that if EL2 is enabled in the current Security state, software cannot access that counter and its associated state from EL0 or EL1.
- I_{SDHCP}** Whether software can access an event counter at an Exception level does not affect whether the counter counts events at that Exception level. For more information, see [Controlling the PMU counters](#) and [Enabling PMU counters](#).

D6.9.1 PMEVCNTR<n> event counters

- D_{YBKKV}** [Table D6-1](#) shows how the [counter range](#) affects the behavior of permitted accesses to the event counter registers.

Table D6-1 Result of event counter System register access

Counter range	Access at Exception level		
	EL2	EL1	EL0
First range	Succeeds	Succeeds	Succeeds
Second range	Succeeds	No access	No access
Counter not implemented	No access	No access	No access

- I_{MWZHC}** Where [Table D6-1](#) shows access succeeds for an event counter *n*, the access might be UNDEFINED or generate a trap exception. See the descriptions of [PMEVCNTR<n>](#) and [PMXEVCNTR](#) for details.
- I_{DXWRB}** Where [Table D6-1](#) shows no access for an event counter *n*, all of the following apply:
- When [PMSELR.SEL](#) is *n*, the PE prevents direct reads and direct writes of [PMXEVTYPER](#) or [PMXEVCNTR](#).
 - The PE prevents direct reads and direct writes of [PMEVTYPER<n>](#) and [PMEVCNTR<n>](#).
 - Direct reads and direct writes of the following register bits are RAZ/WI:
 - [PMOVSLR_EL0\[n\]](#).
 - [PMOVSSET\[n\]](#).
 - [PMCNTENSET\[n\]](#).
 - [PMCNTENCLR\[n\]](#).
 - [PMINTENSET\[n\]](#).
 - [PMINTENCLR\[n\]](#).
 - [PMOVSLR_EL0\[n\]](#).
 - [PMOVSSET\[n\]](#).
 - [PMCNTENSET\[n\]](#).
 - [PMCNTENCLR\[n\]](#).
 - Direct writes to [PMSWINC\[n\]](#) are ignored.
 - A direct write of 1 to [PMCR.P](#) does not reset [PMEVCNTR<n>](#).

D6.9.2 Cycle and instruction counters

The PMU does not provide any control that a hypervisor can use to reserve the cycle counter for its own use. However, access to the PMU registers are subject to the access permissions described in Configurable instruction controls.

D6.9.3 EL0 access controls

- I_{HNKZY}** [PMUSERENR](#) controls EL0 access to the Performance Monitors.

D6.10 Performance Monitors Extension registers

Further information on the Performance Monitors Extension Registers can be found in the following sections:

- [Performance monitors System registers](#) summarizes the Performance Monitors Extension System registers in AArch32 state.
- [Memory-mapped Performance monitors registers](#) summarizes the memory-mapped Performance Monitors Extension registers.

Chapter D7

PMU Event Descriptions

This chapter describes the PMU events that can be counted, their associated event numbers, and the mnemonics for the events. It contains the following sections:

- [Definitions](#)
- [The PMU event number space and common events](#)
- [Common event numbers](#)
- [Cycle event counting](#)
- [Meaningful combinations of common events](#)
- [Required events](#)
- [IMPLEMENTATION DEFINED event numbers](#)

D7.1 Definitions

The following subsections give more information about terms used in the event definitions:

- [Definition of terms.](#)
- [Levels of caches.](#)
- [Counting events from shared components.](#)
- [Counting exceptions taken locally or not taken locally.](#)

D7.1.1 Definition of terms

This section describes terms used by PMU events as they relate to the implementation of the PMU. For more definitions, see [Glossary](#).

Architectural event

An architectural event is an event which gives the same result for the same program on any implementation of the Arm architecture, subject to the program having the same inputs, including asynchronous events triggered by the system, any IMPLEMENTATION DEFINED or UNPREDICTABLE variation permitted by the event definition, and the reasonable degree of inaccuracy described in [A reasonable degree of inaccuracy](#). `INST_RETIRED` is an example of an architectural event.

An architectural event should not be confused with an event defined by the architecture, which is referred to as a Common event.

Note: Do not confuse Architectural events and Common events. An Architectural event can be either Common or IMPLEMENTATION DEFINED. See also Common event and IMPLEMENTATION DEFINED event.

At-retirement event

An at-retirement event is generated by an [Architecturally executed](#) instruction that does not generate an exception, unless specified otherwise. This includes SVC, HVC, and SMC exceptions.

For cache and TLB events:

- The event counts each architecturally-executed instruction that generates the event.
- The event is only counted once, by the first instruction that causes it. This might not be the first instruction in program order.

Note

Consider two loads that execute out of order that access the same cache line granule which is not in the cache. The second instruction in program order causes the miss, and the first instruction fetches data from the same cache miss result. Only the second instruction counts the cache miss event. Also, the second instruction might be speculative and the first instruction retired, meaning the cache miss is not counted at all.

- The population counts for the events are the architecturally-executed instruction counts.

Common event

An event that is defined by the Performance Monitors Extension. For more information, see Common event numbers.

Note: Do not confuse Common events and Architectural events. A Common event can be either Architectural or Microarchitectural. See also Architectural event and Microarchitectural event.

CSIZE

Container size, in bits, that corresponds to the largest non-overlapping Advanced SIMD vector element size or scalar register size that is encoded in the instruction opcode.

Event in progress

Some events count when another event or condition is *in progress*. This might mean that the event counts the occupancy of a queue or other microarchitectural structure tracking the event. It is usually IMPLEMENTATION DEFINED when an event is in progress.

For example, the [MEM_ACCESS_RD_PERCYC](#) event counts when the [MEM_ACCESS_RD](#) event is in progress, meaning on each [Processor cycle](#), the counter increments by the number of Memory-read operations that are in progress.

These events can be used to calculate the average number of events in progress. In the case of the [MEM_ACCESS_RD_PERCYC](#) event, this is also the average read latency. However, the definition of *in progress* might not include all parts of the operation.

Example D7-1

Stages of an operation not included in the definition of event in progress

In an example implementation, a Memory-read operation generated by a load instruction occupies three pipeline stages in the PE before generating a [MEM_ACCESS_RD](#) event when the PE starts to access memory. The event is then considered to be *in progress* until the data is returned to the PE. In the case of a Normal Cacheable access, the PE first looks in the Level 1 data cache, and if the address is cached, returns data in two cycles. Once the data is returned, it is another cycle before the result can be forwarded to any other instruction.

In this example, if all loads hit in the Level 1 cache, the average read latency calculated using the [MEM_ACCESS_RD_PERCYC](#) and [MEM_ACCESS_RD](#) events might be two cycles. Although this value is correct for the IMPLEMENTATION SPECIFIC definition of this event, it has to be adjusted by a constant four additional cycles to match the more commonly understood definition of Level 1 cache access latency, which for this example would be quoted as six cycles.

Instruction architecturally executed

An instruction that is part of the [Execution stream](#). See also [Architecturally executed](#).

A [reasonable degree of inaccuracy](#) allows for counts to be inaccurate in exceptional circumstances. For an event that counts instructions architecturally executed, this allows an implementation to count instructions that do not form part of the Execution stream because of an exceptional event, such as if the instruction generates a synchronous exception or entry to Debug state.

Instructions that have no visible effect on the architectural state of the PE are architecturally executed and counted even if they form part of the [Execution stream](#).

Examples of instructions that have no visible effect are:

- A NOP.
- A conditional instruction that fails its Condition code check.
- A Compare and Branch on Zero, CBZ, instruction that does not branch.
- A Compare and Branch on Nonzero, CBNZ, instruction that does not branch.

However, for events that count only the execution of instructions that update System registers, such as [CID_WRITE_RETIRED](#), if such an instruction is executed twice without an intervening [Context Synchronization event](#), it is constrained unpredictable whether the first instruction is counted.

If FEAT_PMUv3p9 is implemented, then architecturally-executed MOVPRFX instructions are counted.

Otherwise, it is IMPLEMENTATION DEFINED whether architecturally-executed MOVPRFX instructions are counted. This can vary dynamically between MOVPRFX instructions and for each execution of the same MOVPRFX instruction.

Instruction architecturally executed, Condition code check pass

Instruction architecturally executed, Condition code check pass is a class of events that explicitly do not occur for:

- A conditional instruction that fails its Condition code check.
- A Compare and Branch on Zero, CBZ, instruction that does not branch.
- A Compare and Branch on Nonzero, CBNZ, instruction that does not branch.
- A Test and Branch on Zero, TBZ, instruction that does not branch.
- A Test and Branch on Nonzero, TBNZ, instruction that does not branch.

- A Store-Exclusive instruction that does not write to memory.

Otherwise, the definition of architecturally executed is the same as for [Instruction architecturally executed](#).

A branch that is architecturally executed, with condition code check pass is also described as a branch taken.

Instruction memory access

A PE acquires instructions for execution through instruction fetches. Instruction fetches might be due to:

- Fetching instructions that are architecturally executed.
- The result of the execution of an instruction prefetch instruction, `PLI`.
- Speculation that a particular instruction might be executed in the future.

The relationship between the fetch of an individual instruction and an instruction memory access is `IMPLEMENTATION DEFINED`. For example, an implementation might fetch many instructions including a non-integer number of instructions in a single instruction memory access.

IMPLEMENTATION DEFINED event

An event that is defined by the implementation. For more information, see `IMPLEMENTATION DEFINED` event numbers.

Note: Do not confuse `IMPLEMENTATION DEFINED` events and Microarchitectural events. An `IMPLEMENTATION DEFINED` event can be either Architectural or Microarchitectural. See also Architectural event and Microarchitectural event.

Memory-write operations

Memory-write operations might be due to:

- The result of an architecturally executed [Memory-writing instructions](#).
- The result of a [Speculatively executed Memory-writing instructions](#).
- A hardware update of a translation table entry.

Note

[Speculatively executed](#) Memory-writing instructions that do not become architecturally executed must not alter the architecturally defined view of memory. They can, however, generate a Memory-write operation that is later undone in some `IMPLEMENTATION SPECIFIC` way.

For levels of cache hierarchy beyond the Level 1 caches, Memory-write operations also include accesses made as part of a write-back from another cache closer to the PE. Such write-backs might be due to:

- Evicting a dirty line from the cache, to allocate a cache line for a cache refill, see Memory-read operations.
- The execution of a cache maintenance instruction.

Note

A cache maintenance instruction is not in itself an access to that cache. However, it might generate write-backs which are then treated as Memory-write operations beyond that cache.

- The result of a coherency request from another PE.

This list is not exhaustive.

`DC ZVA` is counted as a Memory-write operation.

The relationship between Memory-writing instructions and Memory-write operations is `IMPLEMENTATION DEFINED`. For example, for some implementations an `STP` instruction that writes two 64-bit registers might generate one Memory-write operation if the address is quadword-aligned, but for other addresses it generates two or more Memory-write operations. In some implementations, the result of two `STR` instructions that write to adjacent memory might be merged into a single Memory-write operation.

Note

The data written back from a cache that is shared with other PEs might not be data that was written by the PE that performs the operation that leads to the write-back. Nevertheless, the event is counted as a write-back event for that PE.

Memory-writing instructions

Are instructions with explicit Memory write effects. Stores and atomic instructions are counted as Memory-writing instructions.

Unless otherwise specified, this includes instructions that can have a Memory write effect, even if the instruction does not actually write to memory for some other reason. For example, a CAS or Store-exclusive that does not update memory, or a store that generates an MPU fault.

DC ZVA is counted as Memory-writing instructions.

Microarchitectural event

A microarchitectural event is any event which is not architectural. That is, it will give different results for the same program on two different implementations of the Arm architecture, due to microarchitectural differences in the implementations. [L1D_CACHE_REFILL](#) is an example of a microarchitectural event.

A microarchitectural event should not be confused with an event defined by the implementation, which is referred to as an IMPLEMENTATION DEFINED event.

Note: Do not confuse Microarchitectural events and IMPLEMENTATION DEFINED events. A Microarchitectural event can be either Common or IMPLEMENTATION DEFINED. See also Common event and IMPLEMENTATION DEFINED event.

MSIZE

Memory element access size, in bits, that corresponds to a load or store instruction mnemonic suffix, where B=8, H=16, W=32 and D=64. When an instruction mnemonic does not end with B, H, W or D, the memory access size is implied by the scalar transfer register size or SIMD transfer register element size.

Operation

An event that counts operations counts an IMPLEMENTATION DEFINED choice between:

- Architectural instructions.
- IMPLEMENTATION SPECIFIC microarchitectural operations (micro-ops).

An architectural instruction might create one or more microarchitectural operations (micro-ops) at any point in the execution pipeline. Depending on the event definition, the micro-ops might be counted instead of instructions. The definition of a micro-op is IMPLEMENTATION SPECIFIC. An architecture instruction might create more than one micro-op for each instruction. A micro-op might also be removed or merged with another micro-op in the execution stream, so an architecture instruction might create no micro-ops for an instruction. Any arbitrary translation of instructions to an equivalent sequence of micro-ops is permitted.

Microarchitectural operations might have different characteristics from the architectural instructions they are created from. For events defined with reference to a particular instruction or type of instruction, the PE counts events appropriate to the micro-op.

This means the relationship between a micro-op and an architecturally executed instruction is IMPLEMENTATION DEFINED, and an architecture instruction of one type might be counted as one or more micro-ops of a different type.

However, in order for the profile information to be meaningful to the user of the counter information, Arm recommends that the micro-op is counted as a type appropriate to the instruction when this is appropriate and feasible.

Example D7-2 Counting micro-ops

If an architectural load instruction is split into an address generation micro-op and a load micro-op, and the PMU counts micro-ops, then when counting the operations:

- The instruction is counted as two operations.
- The address generation micro-op is counted as a data-processing operation.
- The load micro-op is counted as a Memory-reading operation.

Example D7-3 Counting SIMD micro-ops

If the PMU counts micro-ops as operations, then a micro-op generated by an SIMD SVE instruction might be counted as an Advanced SIMD operation when all of the following are true:

- The effective vector length for the micro-op is 128 bits.
- The operation is not predicated, and does not have a predicate register as an input or output.
- The operation has an equivalent Advanced SIMD operation.

Operation speculatively executed

An [Operation](#) that is [Speculatively executed](#).

There is no architecturally guaranteed relationship between a [Speculatively executed](#) micro-op and an architecturally executed instruction.

The results of such an operation can also be discarded, if it transpires that the operation was not required, such as following a mispredicted branch. Therefore, the architecture defines these events as *operations speculatively executed*, where appropriate.

Many PMU events count operations that are speculatively executed. This means that the PMU can observe and record the behavior of operations executed under the effects of speculation.

For example:

- An address that is not permitted to be used by the operation might be inferred by whether it is present in a cache.
- A condition code that is not permitted to be used by the operation can be inferred by whether a branch was taken or not taken.

Note

In some events, operation has a more specific meaning described in the event.

Processor cycle

For a non-multithreaded implementation, this means a cycle of the processor. For a multithreaded implementation, processor cycle means each cycle of the multithreaded processor, not just those cycles for which the PE counting the event is active.

Slot

An implementation of a PE might be able to execute multiple micro-ops in a single processor cycle. The maximum number of micro-ops that can be executed might vary at different points in the execution pipeline.

To allow profiling of the utilization of the resource of the PE, an IMPLEMENTATION SPECIFIC point in the execution pipeline is chosen where the maximum number of micro-ops that can be executed is an IMPLEMENTATION DEFINED fixed value.

Each possible micro-op that can be executed at that point in a cycle is called a Slot. The maximum number of micro-ops that can be executed is defined by PMMIR.SLOTS.

Software change of the PC

All of the following are treated as a software change of the PC:

- Branch instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.
- Exception return instructions.

It is IMPLEMENTATION DEFINED whether any or all of the following are treated as software changes of the PC:

- BRK and BKPT instructions.
- An exception generated because an instruction is UNDEFINED.
- The exception-generating instructions, SVC.
- Context synchronization barrier ISB instructions.

Speculatively executed

An instruction or operation that is counted by an event when it might be [Speculative](#).

An operation or instruction might be [Speculative](#), meaning it will be discarded if the operation was incorrectly speculatively executed. For example, following a mispredicted branch. Otherwise, a [Speculative](#) operation or instruction is later retired and becomes [Architecturally executed](#).

The architecture does not define the point in a pipeline where an event is counted. For some events, the event can be counted before the operation or instruction that caused the event is retired. In some cases, this means events might be counted for [Speculative](#) operations or instructions which are later discarded. In other cases, the [Speculative](#) operation or instruction might be discarded before the event is counted, meaning such operations or events are not counted.

Different groups of events might be counted at different points in the pipeline and so can have different IMPLEMENTATION DEFINED definitions of speculatively executed. Such groups share a common base type, which the event name denotes. Each of the events in the previous example is of the base type, operation speculatively executed.

For groups of events with a common base type, speculatively executed operations are all counted on the same basis, which normally means at the same point in the pipeline. It is possible to compare the counts and make meaningful observations about the program being profiled.

Taken locally

Taken locally is a qualifier that determines which instances of an exception are counted by particular PMU events. See [Counting exceptions taken locally or not taken locally](#).

In this context, an exception that is *Taken locally* means an exception that is one of:

- Taken to the current Exception level.

Note

This is not possible when the current Exception level is EL0.

- Taken from EL0 to EL1.
- Taken from EL0 to EL2 because the [Effective value](#) of HCR.TGE is 1.

D7.1.2 Levels of caches

The mapping of levels of cache and TLB to the PMU events is IMPLEMENTATION DEFINED. Although [CLIDR](#) defines the implemented levels of cache, these are not required to correspond with the levels of cache defined for PMU events. The architecture does not provide any way of determining implemented levels of TLB. Also, many implementations include structures that provide some caching at a higher level than the level 1 caches or TLBs. Typically, these structures, that might be called Level 0 caches, or mini caches, or microcaches, are invisible to software. The IMPLEMENTATION SPECIFIC nature of cache and TLB implementations mean that, in general, PMU event counts cannot be used reliably to make direct comparisons between different implementations, and Arm recommends the following implementation guidelines:

- If L3D_CACHE events are implemented, then L2D_CACHE and if applicable L2I_CACHE events should be implemented.
- If L2D_CACHE events are implemented, then L1D_CACHE and if applicable L1I_CACHE events should be implemented.
- If L2I_CACHE events are implemented, then L1I_CACHE events should be implemented.
- Where the Last Level of cache is also the Level 3 or Level 2 unified cache, the LL_CACHE events should be implemented in preference to the L3D_CACHE or L2D_CACHE events as applicable.
- For the Level <n> cache, where <n> = 1 or <n> = 2:
 - If the Level <n> cache is unified, but the cache can disambiguate between Data and Instruction accesses to the cache, then both the L<n>D_CACHE and L<n>I_CACHE events should be implemented.
 - If the cache is unified and the cache cannot disambiguate between Data and Instruction accesses, then only the L<n>D_CACHE should be implemented, counting all accesses.

This final property is IMPLEMENTATION DEFINED.

N1_*, N2_*, N3_*, and N4_* events are defined for identifying the *non-uniform memory access* (NUMA) distance for data returned directly from caches and memory, with support for:

- Up to four NUMA distance levels.
- Up to three cache types at each level.
- Up to three memory types at each level.
- Cache snoop responses at each level.

The NUMA distance indicates the relative distance that the return data has traveled. The NUMA distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have different levels of hierarchy. The interpretation of NUMA distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system.

The mapping of NUMA distance values to LOCAL and REMOTE events is IMPLEMENTATION DEFINED.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be broadly similar for all devices in the system.

D7.1.3 Counting events from shared components

There is no architectural concept of a *shared* component. However, when a cache, a bus, or any other system component that might generate countable events is implemented, and:

- The extent of the first-order effects due to an event from that component are only applicable to a single PE, then the event is not shared.
- Otherwise, the event is shared.

Second-order effects are not considered when determining if an event is shared.

If the cache is shared and the Effective value of `PMEVTYPER<n>.MT` for the counter is `0b0`, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>.MT` for the counter is `0b1`, the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted and might depend on the Effective value of `PMEVTYPER<n>.MT`.

Example D7-4 First and second order effects of a cache miss in a multiple-PE implementation

In an implementation that consists of two PEs, each with its own L1 cache, a cache miss by one of the PEs is a first-order effect of an access to its cache. Any snoop that is performed on the L1 cache of the other PE in the implementation as a result of that cache miss is a second order effect.

Note

Shared events are inherently linked to microarchitectures and so the implementer must make an informed decision about how such events are implemented.

D7.1.4 Counting exceptions taken locally or not taken locally

Table D7-1 shows the events for exceptions taken to an Exception level using AArch32.

Table D7-1 Events for exceptions taken to an Exception level using AArch32

HSR.EC	Description	Event number and classification for exceptions	
		Taken locally	Not Taken locally
0x00	Unknown or uncategorized	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x01	WF* traps	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x03	AArch32 MCR/MRC traps on (coproc == 0b1111) accesses	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x04	AArch32 MCRR/MRRC traps on (coproc == 0b1111) accesses	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x05	AArch32 MCR/MRC traps on (coproc == 0b1110) accesses	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x06	AArch32 LDC/STC traps on (coproc == 0b1110) accesses	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x07	Advanced SIMD or FP traps	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x08	AArch32 MVFR* and FPSID traps	-	0x008D, EXC_TRAP_OTHER
0x0C	AArch32 MCRR/MRRC traps on (coproc == 0b1110) accesses	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x0E	Illegal instruction set state	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x11	AArch32 SVC	0x0082, EXC_SVC	0x008D, EXC_TRAP_OTHER
0x12	AArch32 HVC that is not disabled	-	0x008A, EXC_HVC
0x13	AArch32 SMC that is not disabled to EL2	-	0x008D, EXC_TRAP_OTHER
0x20	Instruction Abort from below	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x21	Instruction Abort from current Exception level	0x0083, EXC_PABORT	-
0x22	PC alignment	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x24	Data Abort from below	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x25	Data Abort from current Exception level	0x0084, EXC_DABORT	-
0x28	AArch32 FP exception	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x30	Breakpoint from below	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x31	Breakpoint from current Exception level	0x0083, EXC_PABORT	-

HSR.EC	Description	Event number and classification for exceptions	
		Taken locally	Not Taken locally
0x34	Watchpoint from below	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x35	Watchpoint from current Exception level	0x0084, EXC_DABORT	-
0x38	AArch32 BKPT instruction	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x3A	AArch32 Vector Catch debug event	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
-	IRQ interrupt	0x0086, EXC_IRQ	0x008E, EXC_TRAP_FIQ
-	FIQ interrupt	0x0087, EXC_FIQ	0x008F, EXC_TRAP_FIQ
All other values	All other exceptions	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER

Note

The [Glossary](#) defines the term [Taken locally](#), that is used in event definitions in [Common microarchitectural events](#). See also Exception levels for more information.

D7.2 The PMU event number space and common events

The event number space is 10 bits. For more information about the entries in the [Allocation](#) column see the text that follows this table:

Table D7-2 Allocation of the PMU event number space

Event numbers	Allocation
0x0000 - 0x003F	Common architectural and microarchitectural events.
0x0040 - 0x00BF	Arm-recommended common architectural and microarchitectural events
0x00C0 - 0x03FF	IMPLEMENTATION DEFINED events.

See PMEVTYPEPER<n>.evtCount for details of the PE behavior when an event number for a reserved or unimplemented PMU event is written to evtCount.

[Table D7-3](#) lists the number, mnemonic, and description of PMU events.

Table D7-3 Event index

Number	Mnemonic	Description
0x0000	SW_INCR	Instruction architecturally executed, Condition code check pass, software increment.
0x0001	L1I_CACHE_REFILL	Level 1 instruction cache refill.
0x0002	L1I_TLB_REFILL	Level 1 instruction TLB refill.
0x0003	L1D_CACHE_REFILL	Level 1 data cache refill.
0x0004	L1D_CACHE	Level 1 data cache access.
0x0005	L1D_TLB_REFILL	Level 1 data TLB refill.
0x0006	LD_RETIRED	Instruction architecturally executed, Condition code check pass, load.
0x0007	ST_RETIRED	Instruction architecturally executed, Condition code check pass, store.
0x0008	INST_RETIRED	Instruction architecturally executed.
0x0009	EXC_TAKEN	Exception taken.
0x000A	EXC_RETURN	Instruction architecturally executed, Condition code check pass, exception return.
0x000B	CID_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR.
0x000C	PC_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, Software change of the PC.
0x000D	BR_IMMED_RETIRED	Branch instruction architecturally executed, immediate.

Number	Mnemonic	Description
0x000E	BR_RETURN_RETIRED	Branch instruction architecturally executed, procedure return, taken.
0x000F	UNALIGNED_LDST_RETIRED	Instruction architecturally executed, Condition code check pass, unaligned load or store.
0x0010	BR_MIS_PRED	Branch instruction speculatively executed, mispredicted or not predicted.
0x0011	CPU_CYCLES	Cycle.
0x0012	BR_PRED	Predictable branch instruction speculatively executed.
0x0013	MEM_ACCESS	Data memory access.
0x0014	L1I_CACHE	Level 1 instruction cache access.
0x0015	L1D_CACHE_WB	Level 1 data cache write-back.
0x0016	L2D_CACHE	Level 2 data cache access.
0x0017	L2D_CACHE_REFILL	Level 2 data cache refill.
0x0018	L2D_CACHE_WB	Level 2 data cache write-back.
0x0019	BUS_ACCESS	Bus access.
0x001A	MEMORY_ERROR	Local memory error.
0x001B	INST_SPEC	Operation speculatively executed.
0x001C	TTBR_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, write to TTBR.
0x001D	BUS_CYCLES	Bus cycle.
0x001E	CHAIN	Chain a pair of event counters.
0x001F	L1D_CACHE_ALLOCATE	Level 1 data cache allocation without refill.
0x0020	L2D_CACHE_ALLOCATE	Level 2 data cache allocation without refill.
0x0021	BR_RETIRED	Instruction architecturally executed, branch.
0x0022	BR_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted.
0x0023	STALL_FRONTEND	No operation sent for execution due to the frontend.
0x0024	STALL_BACKEND	No operation sent for execution due to the backend.
0x0025	L1D_TLB	Level 1 data TLB access.
0x0026	L1I_TLB	Level 1 instruction TLB access.
0x0027	L2I_CACHE	Level 2 instruction cache access.
0x0028	L2I_CACHE_REFILL	Level 2 instruction cache refill.
0x0029	L3D_CACHE_ALLOCATE	Level 3 data cache allocation without refill.
0x002A	L3D_CACHE_REFILL	Level 3 data cache refill.
0x002B	L3D_CACHE	Level 3 data cache access.

Number	Mnemonic	Description
0x002C	L3D_CACHE_WB	Level 3 data cache write-back.
0x002D	L2D_TLB_REFILL	Level 2 data TLB refill.
0x002E	L2I_TLB_REFILL	Level 2 instruction TLB refill.
0x002F	L2D_TLB	Level 2 data TLB access.
0x0030	L2I_TLB	Level 2 instruction TLB access.
0x0031	REMOTE_ACCESS	Access to a remote device.
0x0032	LL_CACHE	Last level cache access.
0x0033	LL_CACHE_MISS	Last level cache miss.
0x0034	DTLB_WALK	Data TLB access with at least one translation table walk.
0x0035	ITLB_WALK	Instruction TLB access with at least one translation table walk.
0x0036	LL_CACHE_RD	Last level cache access, read.
0x0037	LL_CACHE_MISS_RD	Last level cache miss, read.
0x0038	REMOTE_ACCESS_RD	Access to a remote device, read.
0x0039	L1D_CACHE_LMISS_RD	Level 1 data cache long-latency read miss.
0x003A	OP_RETIRED	Micro-operation architecturally executed.
0x003B	OP_SPEC	Micro-operation speculatively executed.
0x003C	STALL	No operation sent for execution.
0x003D	STALL_SLOT_BACKEND	No operation sent for execution on a Slot due to the backend.
0x003E	STALL_SLOT_FRONTEND	No operation sent for execution on a Slot due to the frontend.
0x003F	STALL_SLOT	No operation sent for execution on a Slot.
0x0040	L1D_CACHE_RD	Level 1 data cache access, read.
0x0041	L1D_CACHE_WR	Level 1 data cache access, write.
0x0042	L1D_CACHE_REFILL_RD	Level 1 data cache refill, read.
0x0043	L1D_CACHE_REFILL_WR	Level 1 data cache refill, write.
0x0044	L1D_CACHE_REFILL_INNER	Level 1 data cache refill, inner.
0x0045	L1D_CACHE_REFILL_OUTER	Level 1 data cache refill, outer.
0x0046	L1D_CACHE_WB_VICTIM	Level 1 data cache write-back, victim.
0x0047	L1D_CACHE_WB_CLEAN	Level 1 data cache write-back, cleaning and coherency.
0x0048	L1D_CACHE_INVALID	Level 1 data cache invalidate.
0x004C	L1D_TLB_REFILL_RD	Level 1 data TLB refill, read.
0x004D	L1D_TLB_REFILL_WR	Level 1 data TLB refill, write.
0x004E	L1D_TLB_RD	Level 1 data TLB access, read.
0x004F	L1D_TLB_WR	Level 1 data TLB access, write.
0x0050	L2D_CACHE_RD	Level 2 data cache access, read.

Number	Mnemonic	Description
0x0051	L2D_CACHE_WR	Level 2 data cache access, write.
0x0052	L2D_CACHE_REFILL_RD	Level 2 data cache refill, read.
0x0053	L2D_CACHE_REFILL_WR	Level 2 data cache refill, write.
0x0056	L2D_CACHE_WB_VICTIM	Level 2 data cache write-back, victim.
0x0057	L2D_CACHE_WB_CLEAN	Level 2 data cache write-back, cleaning and coherency.
0x0058	L2D_CACHE_INVALID	Level 2 data cache invalidate.
0x005C	L2D_TLB_REFILL_RD	Level 2 data TLB refill, read.
0x005D	L2D_TLB_REFILL_WR	Level 2 data TLB refill, write.
0x005E	L2D_TLB_RD	Level 2 data TLB access, read.
0x005F	L2D_TLB_WR	Level 2 data TLB access, write.
0x0060	BUS_ACCESS_RD	Bus access, read.
0x0061	BUS_ACCESS_WR	Bus access, write.
0x0062	BUS_ACCESS_SHARED	Bus access, Normal, Cacheable, Shareable.
0x0063	BUS_ACCESS_NOT_SHARED	Bus access, not Normal, Cacheable, Shareable.
0x0064	BUS_ACCESS_NORMAL	Bus access, normal.
0x0065	BUS_ACCESS_PERIPH	Bus access, peripheral.
0x0066	MEM_ACCESS_RD	Data memory access, read.
0x0067	MEM_ACCESS_WR	Data memory access, write.
0x0068	UNALIGNED_LD_SPEC	Unaligned access, read.
0x0069	UNALIGNED_ST_SPEC	Unaligned access, write.
0x006A	UNALIGNED_LDST_SPEC	Unaligned access.
0x006C	LDREX_SPEC	Exclusive operation speculatively executed, Load-Exclusive.
0x006D	STREX_PASS_SPEC	Exclusive operation speculatively executed, Store-Exclusive pass.
0x006E	STREX_FAIL_SPEC	Exclusive operation speculatively executed, Store-Exclusive fail.
0x006F	STREX_SPEC	Exclusive operation speculatively executed, Store-Exclusive.
0x0070	LD_SPEC	Operation speculatively executed, load.
0x0071	ST_SPEC	Operation speculatively executed, store.
0x0072	LDST_SPEC	Operation speculatively executed, load or store.
0x0073	DP_SPEC	Operation speculatively executed, integer data processing.
0x0074	ASE_SPEC	Operation speculatively executed, Advanced SIMD data processing.
0x0075	VFP_SPEC	Operation speculatively executed, scalar floating-point data processing.

Number	Mnemonic	Description
0x0076	PC_WRITE_SPEC	Operation speculatively executed, Software change of the PC.
0x0077	CRYPTO_SPEC	Operation speculatively executed, cryptographic data processing.
0x0078	BR_IMMED_SPEC	Branch speculatively executed, immediate branch.
0x0079	BR_RETURN_SPEC	Branch speculatively executed, procedure return.
0x007A	BR_INDIRECT_SPEC	Branch speculatively executed, indirect branch.
0x007C	ISB_SPEC	Barrier speculatively executed, ISB.
0x007D	DSB_SPEC	Barrier speculatively executed, DSB.
0x007E	DMB_SPEC	Barrier speculatively executed, DMB.
0x007F	CSDB_SPEC	Barrier speculatively executed, CSDB.
0x0081	EXC_UNDEF	Exception taken, other synchronous.
0x0082	EXC_SVC	Exception taken, Supervisor Call.
0x0083	EXC_PABORT	Exception taken, Instruction Abort.
0x0084	EXC_DABORT	Exception taken, Data Abort or SError.
0x0086	EXC_IRQ	Exception taken, IRQ.
0x0087	EXC_FIQ	Exception taken, FIQ.
0x0088	EXC_SMC	Exception taken, Secure Monitor Call.
0x008A	EXC_HVC	Exception taken, Hypervisor Call.
0x008B	EXC_TRAP_PABORT	Exception taken, Instruction Abort not Taken locally.
0x008C	EXC_TRAP_DABORT	Exception taken, Data Abort or SError not Taken locally.
0x008D	EXC_TRAP_OTHER	Exception taken, other traps not Taken locally.
0x008E	EXC_TRAP_IRQ	Exception taken, IRQ not Taken locally.
0x008F	EXC_TRAP_FIQ	Exception taken, FIQ not Taken locally.
0x0090	RC_LD_SPEC	Release consistency operation speculatively executed, Load-Acquire.
0x0091	RC_ST_SPEC	Release consistency operation speculatively executed, Store-Release.
0x00A0	L3D_CACHE_RD	Level 3 data cache access, read.
0x00A1	L3D_CACHE_WR	Level 3 data cache access, write.
0x00A2	L3D_CACHE_REFILL_RD	Level 3 data cache refill, read.
0x00A3	L3D_CACHE_REFILL_WR	Level 3 data cache refill, write.
0x00A6	L3D_CACHE_WB_VICTIM	Level 3 data cache write-back, victim.

Number	Mnemonic	Description
0x00A7	L3D_CACHE_WB_CLEAN	Level 3 data cache write-back, cleaning and coherency.
0x00A8	L3D_CACHE_INVALID	Level 3 data cache invalidate.
	ASE_FP_ADDSUB_SPEC	Floating-point operation speculatively executed, Advanced SIMD add or subtract.
	ASE_FP_BF16_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is BFloat16 floating-point.
	ASE_FP_CVT_SPEC	Floating-point operation speculatively executed, Advanced SIMD convert.
	ASE_FP_DIV_SPEC	Floating-point operation speculatively executed, Advanced SIMD divide.
	ASE_FP_DOT_SPEC	Floating-point operation speculatively executed, Advanced SIMD dot-product.
	ASE_FP_DP_SPEC	Floating-point operation speculatively executed, Advanced SIMD double-precision, data-processing.
	ASE_FP_FMA_SPEC	Floating-point operation speculatively executed, Advanced SIMD FMA.
	ASE_FP_FP8_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is 8-bit floating-point.
	ASE_FP_HP_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is half-precision floating-point.
	ASE_FP_HP_SPEC	Floating-point operation speculatively executed, Advanced SIMD half-precision, data-processing.
	ASE_FP_MMLA_SPEC	Floating-point operation speculatively executed, Advanced SIMD matrix multiply.
	ASE_FP_MUL_SPEC	Floating-point operation speculatively executed, Advanced SIMD multiply.
	ASE_FP_PREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD pairwise add step or pairwise reduce step.
	ASE_FP_RECPE_SPEC	Floating-point operation speculatively executed, Advanced SIMD reciprocal estimate.
	ASE_FP_SPEC	Floating-point operation speculatively executed, Advanced SIMD.
	ASE_FP_SP_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is single-precision floating-point.

Number	Mnemonic	Description
	ASE_FP_SP_SPEC	Floating-point operation speculatively executed, Advanced SIMD single-precision, data-processing.
	ASE_FP_SQRT_SPEC	Floating-point operation speculatively executed, Advanced SIMD square root.
	ASE_FP_VREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD pairwise or reduction.
	ASE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD.
	ASE_INST_SPEC	Operation speculatively executed, Advanced SIMD.
	ASE_INT16_SPEC	Integer operation speculatively executed, Advanced SIMD 16-bit.
	ASE_INT32_SPEC	Integer operation speculatively executed, Advanced SIMD 32-bit.
	ASE_INT64_SPEC	Integer operation speculatively executed, Advanced SIMD 64-bit.
	ASE_INT8_SPEC	Integer operation speculatively executed, Advanced SIMD 8-bit.
	ASE_INT_DOT_SPEC	Operation speculatively executed, Advanced SIMD integer dot-product.
	ASE_INT_MMLA_SPEC	Integer operation speculatively executed, Advanced SIMD matrix multiply.
	ASE_INT_MUL_SPEC	Integer operation speculatively executed, Advanced SIMD multiply.
	ASE_INT_SPEC	Integer operation speculatively executed, Advanced SIMD.
	ASE_INT_VREDUCE_SPEC	Integer operation speculatively executed, Advanced SIMD reduction.
	ASE_LUT_SPEC	Lookup table operation speculatively executed, Advanced SIMD data processing.
	ASE_NONFP_SPEC	Non-floating-point operation speculatively executed, Advanced SIMD.
	ASE_RETIRED	Instruction architecturally executed, Advanced SIMD data processing.
	ASE_SVE_FP_ADDSUB_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE add or subtract .
	ASE_SVE_FP_BF16_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point.
	ASE_SVE_FP_BF16_SPEC	Floating-point operation speculatively executed, SVE or Advanced SIMD BFloat16, data-processing.

Number	Mnemonic	Description
	ASE_SVE_FP_CVT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE convert.
	ASE_SVE_FP_DIV_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE divide.
	ASE_SVE_FP_DOT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE dot-product.
	ASE_SVE_FP_DP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE double-precision, data-processing.
	ASE_SVE_FP_FMA_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE FMA.
	ASE_SVE_FP_FP8_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is 8-bit floating-point.
	ASE_SVE_FP_HP_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is half-precision floating-point.
	ASE_SVE_FP_HP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE half-precision.
	ASE_SVE_FP_MMLA_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE matrix multiply.
	ASE_SVE_FP_MUL_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE multiply.
	ASE_SVE_FP_RECPE_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE reciprocal estimate.
	ASE_SVE_FP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE.
	ASE_SVE_FP_SP_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is single-precision floating-point.
	ASE_SVE_FP_SP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE single-precision, data-processing.
	ASE_SVE_FP_SQRT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE square-root.
	ASE_SVE_FP_VREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE vector reduction.
	ASE_SVE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD or SVE.

Number	Mnemonic	Description
	ASE_SVE_INST_SPEC	Operation speculatively executed, Advanced SIMD or SVE.
	ASE_SVE_INT16_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 16-bit.
	ASE_SVE_INT32_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 32-bit.
	ASE_SVE_INT64_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 64-bit.
	ASE_SVE_INT8_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 8-bit.
	ASE_SVE_INT_DOT_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE dot-product.
	ASE_SVE_INT_MMLA_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE matrix multiply.
	ASE_SVE_INT_MUL_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE multiply.
	ASE_SVE_INT_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE.
	ASE_SVE_INT_VREDUCE_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE reduction.
	ASE_SVE_LDST_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous load or store multiple vector.
	ASE_SVE_LDST_SPEC	Operation speculatively executed, Advanced SIMD or SVE load or store.
	ASE_SVE_LD_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous load multiple vector.
	ASE_SVE_LD_SPEC	Operation speculatively executed, Advanced SIMD or SVE load.
	ASE_SVE_NONFP_SPEC	Non-floating-point operation speculatively executed, Advanced SIMD or SVE.
	ASE_SVE_RETIRED	Instruction architecturally executed, Advanced SIMD data processing or SVE data processing.
	ASE_SVE_SPEC	Operation speculatively executed, Advanced SIMD data processing or SVE data processing.
	ASE_SVE_ST_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous store multiple vector.

Number	Mnemonic	Description
	ASE_SVE_ST_SPEC	Operation speculatively executed, Advanced SIMD or SVE store.
	ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC	Advanced SIMD or SVE unaligned contiguous access.
	ASE_SVE_UNALIGNED_CONTIG_LD_SPEC	Advanced SIMD or SVE unaligned contiguous read.
	ASE_SVE_UNALIGNED_CONTIG_ST_SPEC	Advanced SIMD or SVE unaligned contiguous write.
	ASE_SVE_UNALIGNED_LDST_SPEC	Advanced SIMD or SVE unaligned access.
	ASE_SVE_UNALIGNED_LD_SPEC	Advanced SIMD or SVE unaligned read.
	ASE_SVE_UNALIGNED_ST_SPEC	Advanced SIMD or SVE unaligned write.
	ASE_SVE_UOP_SPEC	Microarchitectural operation speculatively executed, Advanced SIMD or SVE.
	ASE_UOP_SPEC	Microarchitectural operation speculatively executed, Advanced SIMD.
	BASE_LDST_REG_SPEC	Operation speculatively executed, general-purpose register load, store, or prefetch.
	BASE_LD_REG_SPEC	Operation speculatively executed, general-purpose register load.
	BASE_PRF_SPEC	Operation speculatively executed, general-purpose register prefetch.
	BASE_ST_REG_SPEC	Operation speculatively executed, general-purpose register store.
	BL_IMMED_TAKEN_RETIRED	Branch instruction architecturally executed, direct branch with link, taken.
	BL_IND_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch with link, taken.
	BL_TAKEN_RETIRED	Branch instruction architecturally executed, branch with link, taken.
	BRB_FILTRATE	Branch Record captured.
	BRNL_IMMED_TAKEN_RETIRED	Branch instruction architecturally executed, direct branch without link, taken.
	BRNL_INDNR_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch without link excluding procedure return, taken.
	BRNL_IND_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch without link, taken.
	BRNL_TAKEN_RETIRED	Branch instruction architecturally executed, branch without link, taken.

Number	Mnemonic	Description
	BR_COND_RETIRE	Branch instruction architecturally executed, conditional branch.
	BR_COND_TAKEN_RETIRE	Branch instruction architecturally executed, conditional branch, taken.
	BR_IMMED_MIS_PRED_RETIRE	Branch instruction architecturally executed, mispredicted immediate.
	BR_IMMED_PRED_RETIRE	Branch instruction architecturally executed, predicted immediate.
	BR_IMMED_SKIP_RETIRE	Branch instruction architecturally executed, immediate, not taken.
	BR_IMMED_TAKEN_RETIRE	Branch instruction architecturally executed, immediate, taken.
	BR_INDNR_MIS_PRED_RETIRE	Branch instruction architecturally executed, mispredicted indirect excluding procedure return.
	BR_INDNR_PRED_RETIRE	Branch instruction architecturally executed, predicted indirect excluding procedure return.
	BR_INDNR_RETIRE	Branch instruction architecturally executed, indirect excluding procedure return.
	BR_INDNR_SKIP_RETIRE	Branch instruction architecturally executed, indirect excluding procedure return, not taken.
	BR_INDNR_TAKEN_RETIRE	Branch instruction architecturally executed, indirect excluding procedure return, taken.
	BR_IND_MIS_PRED_RETIRE	Branch instruction architecturally executed, mispredicted indirect.
	BR_IND_PRED_RETIRE	Branch instruction architecturally executed, predicted indirect.
	BR_IND_RETIRE	Instruction architecturally executed, indirect branch.
	BR_IND_SKIP_RETIRE	Branch instruction architecturally executed, indirect, not taken.
	BR_IND_TAKEN_RETIRE	Branch instruction architecturally executed, indirect, taken.
	BR_PRED_RETIRE	Branch instruction architecturally executed, predicted branch.
	BR_RETURN_ANY_RETIRE	Branch instruction architecturally executed, procedure return.
	BR_RETURN_MIS_PRED_RETIRE	Branch instruction architecturally executed, mispredicted procedure return.
	BR_RETURN_PRED_RETIRE	Branch instruction architecturally executed, predicted procedure return.
	BR_RETURN_SKIP_RETIRE	Branch instruction architecturally executed, procedure return, not taken.

Number	Mnemonic	Description
	BR_SKIP_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted branch, not taken.
	BR_SKIP_PRED_RETIRED	Branch instruction architecturally executed, predicted branch, not taken.
	BR_SKIP_RETIRED	Branch instruction architecturally executed, not taken.
	BR_TAKEN_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted branch, taken.
	BR_TAKEN_PRED_RETIRED	Branch instruction architecturally executed, predicted branch, taken.
	BR_UNCOND_RETIRED	Branch instruction architecturally executed, unconditional branch.
	BUS_REQ	Bus request.
	BUS_REQ_RD	Bus request, read.
	BUS_REQ_RD_PERCYC	Bus read transactions in progress.
	BUS_REQ_WR	Bus request, write.
	BUS_REQ_WR_PERCYC	Bus write transactions in progress.
	CNT_CYCLES	Constant frequency cycles.
	CRYPTO_RETIRED	Instruction architecturally executed, cryptographic data processing.
	CTI_TRIGOUT4	Cross-trigger Interface output trigger 4.
	CTI_TRIGOUT5	Cross-trigger Interface output trigger 5.
	CTI_TRIGOUT6	Cross-trigger Interface output trigger 6.
	CTI_TRIGOUT7	Cross-trigger Interface output trigger 7.
	DP_RETIRED	Instruction architecturally executed, integer data processing.
	DSNP_HIT	Snoop hit, data.
	DSNP_HITM_FAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in far cache.
	DSNP_HITM_FAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in far cache.
	DSNP_HITM_FAR_RD	Snoop hit in far local cache, demand data read, modified.
	DSNP_HITM_FAR_RW	Snoop hit in far local cache, demand data access, modified.
	DSNP_HITM_FAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in far cache.
	DSNP_HITM_FAR_WR	Snoop hit in far local cache, demand data write, modified.
	DSNP_HITM_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit.

Number	Mnemonic	Description
	DSNP_HITM_LD_RETIRED	Load instruction architecturally executed, snoop hit.
	DSNP_HITM_N1_RD	Snoop hit in cache at distance 1, demand data read, modified.
	DSNP_HITM_N1_RW	Snoop hit in cache at distance 1, demand data access, modified.
	DSNP_HITM_N1_WR	Snoop hit in cache at distance 1, demand data write, modified.
	DSNP_HITM_N2_RD	Snoop hit in cache at distance 2, demand data read, modified.
	DSNP_HITM_N2_RW	Snoop hit in cache at distance 2, demand data access, modified.
	DSNP_HITM_N2_WR	Snoop hit in cache at distance 2, demand data write, modified.
	DSNP_HITM_N3_RD	Snoop hit in cache at distance 3, demand data read, modified.
	DSNP_HITM_N3_RW	Snoop hit in cache at distance 3, demand data access, modified.
	DSNP_HITM_N3_WR	Snoop hit in cache at distance 3, demand data write, modified.
	DSNP_HITM_N4_RD	Snoop hit in cache at distance 4, demand data read, modified.
	DSNP_HITM_N4_RW	Snoop hit in cache at distance 4, demand data access, modified.
	DSNP_HITM_N4_WR	Snoop hit in cache at distance 4, demand data write, modified.
	DSNP_HITM_NEAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in near cache.
	DSNP_HITM_NEAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in near cache.
	DSNP_HITM_NEAR_RD	Snoop hit in near local cache, demand data read, modified.
	DSNP_HITM_NEAR_RW	Snoop hit in near local cache, demand data access, modified.
	DSNP_HITM_NEAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in near cache.
	DSNP_HITM_NEAR_WR	Snoop hit in near local cache, demand data write, modified.
	DSNP_HITM_RD	Snoop hit, demand data read, modified.
	DSNP_HITM_REMOTE_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in remote cache.
	DSNP_HITM_REMOTE_LD_RETIRED	Load instruction architecturally executed, snoop hit in remote cache.
	DSNP_HITM_REMOTE_RD	Snoop hit in remote cache, demand data read, modified.

Number	Mnemonic	Description
	DSNP_HITM_REMOTE_RW	Snoop hit in remote cache, demand data access, modified.
	DSNP_HITM_REMOTE_ST_RETIRED	Store instruction architecturally executed, snoop hit in remote cache.
	DSNP_HITM_REMOTE_WR	Snoop hit in remote cache, demand data write, modified.
	DSNP_HITM_RW	Snoop hit, demand data access, modified.
	DSNP_HITM_ST_RETIRED	Store instruction architecturally executed, snoop hit.
	DSNP_HITM_WR	Snoop hit, demand data write, modified.
	DSNP_HIT_FAR	Snoop hit in far local cache, data access.
	DSNP_HIT_FAR_HWPRF	Snoop hit in far local cache, hardware data prefetch.
	DSNP_HIT_FAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in far cache.
	DSNP_HIT_FAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in far cache.
	DSNP_HIT_FAR_PRF	Snoop hit in far local cache, data prefetch.
	DSNP_HIT_FAR_PRFM	Snoop hit in far local cache, software data prefetch.
	DSNP_HIT_FAR_RD	Snoop hit in far local cache, demand data read.
	DSNP_HIT_FAR_RW	Snoop hit in far local cache, demand data access.
	DSNP_HIT_FAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in far cache.
	DSNP_HIT_FAR_WR	Snoop hit in far local cache, demand data write.
	DSNP_HIT_HWPRF	Snoop hit, hardware data prefetch.
	DSNP_HIT_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit.
	DSNP_HIT_LD_RETIRED	Load instruction architecturally executed, snoop hit.
	DSNP_HIT_N1	Snoop hit in cache at distance 1, data access.
	DSNP_HIT_N1_HWPRF	Snoop hit in cache at distance 1, hardware data prefetch.
	DSNP_HIT_N1_PRF	Snoop hit in cache at distance 1, data prefetch.
	DSNP_HIT_N1_PRFM	Snoop hit in cache at distance 1, software data prefetch.
	DSNP_HIT_N1_RD	Snoop hit in cache at distance 1, demand data read.

Number	Mnemonic	Description
	DSNP_HIT_N1_RW	Snoop hit in cache at distance 1, demand data access.
	DSNP_HIT_N1_WR	Snoop hit in cache at distance 1, demand data write.
	DSNP_HIT_N2	Snoop hit in cache at distance 2, data access.
	DSNP_HIT_N2_HWPRF	Snoop hit in cache at distance 2, hardware data prefetch.
	DSNP_HIT_N2_PRF	Snoop hit in cache at distance 2, data prefetch.
	DSNP_HIT_N2_PRFM	Snoop hit in cache at distance 2, software data prefetch.
	DSNP_HIT_N2_RD	Snoop hit in cache at distance 2, demand data read.
	DSNP_HIT_N2_RW	Snoop hit in cache at distance 2, demand data access.
	DSNP_HIT_N2_WR	Snoop hit in cache at distance 2, demand data write.
	DSNP_HIT_N3	Snoop hit in cache at distance 3, data access.
	DSNP_HIT_N3_HWPRF	Snoop hit in cache at distance 3, hardware data prefetch.
	DSNP_HIT_N3_PRF	Snoop hit in cache at distance 3, data prefetch.
	DSNP_HIT_N3_PRFM	Snoop hit in cache at distance 3, software data prefetch.
	DSNP_HIT_N3_RD	Snoop hit in cache at distance 3, demand data read.
	DSNP_HIT_N3_RW	Snoop hit in cache at distance 3, demand data access.
	DSNP_HIT_N3_WR	Snoop hit in cache at distance 3, demand data write.
	DSNP_HIT_N4	Snoop hit in cache at distance 4, data access.
	DSNP_HIT_N4_HWPRF	Snoop hit in cache at distance 4, hardware data prefetch.
	DSNP_HIT_N4_PRF	Snoop hit in cache at distance 4, data prefetch.
	DSNP_HIT_N4_PRFM	Snoop hit in cache at distance 4, software data prefetch.
	DSNP_HIT_N4_RD	Snoop hit in cache at distance 4, demand data read.
	DSNP_HIT_N4_RW	Snoop hit in cache at distance 4, demand data access.
	DSNP_HIT_N4_WR	Snoop hit in cache at distance 4, demand data write.
	DSNP_HIT_NEAR	Snoop hit in near local cache, data access.

Number	Mnemonic	Description
	DSNP_HIT_NEAR_HWPRF	Snoop hit in near local cache, hardware data prefetch.
	DSNP_HIT_NEAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in near cache.
	DSNP_HIT_NEAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in near cache.
	DSNP_HIT_NEAR_PRF	Snoop hit in near local cache, data prefetch.
	DSNP_HIT_NEAR_PRFM	Snoop hit in near local cache, software data prefetch.
	DSNP_HIT_NEAR_RD	Snoop hit in near local cache, demand data read.
	DSNP_HIT_NEAR_RW	Snoop hit in near local cache, demand data access.
	DSNP_HIT_NEAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in near cache.
	DSNP_HIT_NEAR_WR	Snoop hit in near local cache, demand data write.
	DSNP_HIT_PRF	Snoop hit, data prefetch.
	DSNP_HIT_PRFM	Snoop hit, software data prefetch.
	DSNP_HIT_RD	Snoop hit, demand data read.
	DSNP_HIT_REMOTE	Snoop hit in remote cache, data access.
	DSNP_HIT_REMOTE_HWPRF	Snoop hit in remote cache, hardware data prefetch.
	DSNP_HIT_REMOTE_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in remote cache.
	DSNP_HIT_REMOTE_LD_RETIRED	Load instruction architecturally executed, snoop hit in remote cache.
	DSNP_HIT_REMOTE_PRF	Snoop hit in remote cache, data prefetch.
	DSNP_HIT_REMOTE_PRFM	Snoop hit in remote cache, software data prefetch.
	DSNP_HIT_REMOTE_RD	Snoop hit in remote cache, demand data read.
	DSNP_HIT_REMOTE_RW	Snoop hit in remote cache, demand data access.
	DSNP_HIT_REMOTE_ST_RETIRED	Store instruction architecturally executed, snoop hit in remote cache.
	DSNP_HIT_REMOTE_WR	Snoop hit in remote cache, demand data write.
	DSNP_HIT_RW	Snoop hit, demand data access.
	DSNP_HIT_ST_RETIRED	Store instruction architecturally executed, snoop hit.
	DSNP_HIT_WR	Snoop hit, demand data write.

Number	Mnemonic	Description
	DTLB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, no translation table walk.
	DTLB_HIT_LD_RETIRED	Load instruction architecturally executed, no translation table walk.
	DTLB_HIT_ST_RETIRED	Store instruction architecturally executed, no translation table walk.
	DTLB_WALK_HWPRF	Data TLB hardware prefetch, with at least one translation table walk.
	DTLB_WALK_LDST_RETIRED	Load or store instruction architecturally executed, at least one translation table walk.
	DTLB_WALK_LD_RETIRED	Load instruction architecturally executed, at least one translation table walk.
	DTLB_WALK_PERCYC	Data translation table walks in progress.
	DTLB_WALK_PRF	Data TLB prefetch, with at least one translation table walk.
	DTLB_WALK_PRFM	Data TLB software prefetch, with at least one translation table walk.
	DTLB_WALK_RW	Data TLB demand access, with at least one translation table walk.
	DTLB_WALK_ST_RETIRED	Store instruction architecturally executed, at least one translation table walk.
	FPASE_LDST_REG_SPEC	Operation speculatively executed, SIMD&FP register load or store.
	FPASE_LD_REG_SPEC	Operation speculatively executed, SIMD&FP register load.
	FPASE_ST_REG_SPEC	Operation speculatively executed, SIMD&FP register store.
	FP_ADDSUB_SPEC	Floating-point operation speculatively executed, add or subtract.
	FP_BF16_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is BFloat16 floating-point.
	FP_BF16_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is BFloat16 floating-point.
	FP_BF16_MIN_SPEC	Floating-point operation speculatively executed, smallest type is BFloat16 floating-point.
	FP_CVT_SPEC	Floating-point operation speculatively executed, convert.
	FP_DIV_SPEC	Floating-point operation speculatively executed, divide.

Number	Mnemonic	Description
	FP_DP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is double-precision floating-point.
	FP_DP_SPEC	Floating-point operation speculatively executed, double-precision, data-processing.
	FP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, floating-point.
	FP_FMA_SPEC	Floating-point operation speculatively executed, FMA.
	FP_FP8_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is 8-bit floating-point.
	FP_FP8_MIN_SPEC	Floating-point operation speculatively executed, smallest type is 8-bit floating-point.
	FP_HP_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is half-precision floating-point.
	FP_HP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is half-precision floating-point.
	FP_HP_MIN_SPEC	Floating-point operation speculatively executed, smallest type is half-precision floating-point.
	FP_HP_SPEC	Floating-point operation speculatively executed, half-precision.
	FP_MUL_SPEC	Floating-point operation speculatively executed, multiply.
	FP_RECPE_SPEC	Floating-point operation speculatively executed, reciprocal estimate.
	FP_SPEC	Floating-point operation speculatively executed, including SIMD.
	FP_SP_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is single-precision floating-point.
	FP_SP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is single-precision floating-point.
	FP_SP_MIN_SPEC	Floating-point operation speculatively executed, smallest type is single-precision floating-point.
	FP_SP_SPEC	Floating-point operation speculatively executed, single-precision.
	FP_SQRT_SPEC	Floating-point operation speculatively executed, square root.

Number	Mnemonic	Description
	GCSTLB_WALK	GCS TLB access with at least one translation table walk.
	GCSTLB_WALK_PERCYC	Translation table walks in progress.
	GCSTLB_WALK_RW	GCS TLB demand access with at least one translation table walk.
	GCSTLB_WALK_STEP	GCS TLB translation table walk, step.
	INST_FETCH	Instruction memory access.
	INST_FETCH_PERCYC	Instruction fetches in progress.
	INST_FETCH_PRFM	Instruction memory access, software prefetch.
	INST_FETCH_RD	Instruction memory access, demand fetch.
	INT_DIV64_SPEC	Integer operation speculatively executed, 64-bit divide.
	INT_DIV_SPEC	Integer operation speculatively executed, divide.
	INT_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, integer.
	INT_MUL64_SPEC	Integer operation speculatively executed, 64×64 multiply.
	INT_MULH64_SPEC	Integer operation speculatively executed, 64×64 multiply returning high part.
	INT_MUL_SPEC	Integer operation speculatively executed, multiply.
	INT_SPEC	Integer operation speculatively executed.
	ISNP_HIT	Snoop hit, instruction.
	ISNP_HIT_FAR	Snoop hit in far local cache, instruction access.
	ISNP_HIT_FAR_HWPRF	Snoop hit in far local cache, hardware instruction prefetch.
	ISNP_HIT_FAR_PRF	Snoop hit in far local cache, instruction prefetch.
	ISNP_HIT_FAR_PRFM	Snoop hit in far local cache, software instruction prefetch.
	ISNP_HIT_FAR_RD	Snoop hit in far local cache, demand instruction fetch.
	ISNP_HIT_FAR_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in far cache.
	ISNP_HIT_HWPRF	Snoop hit, hardware instruction prefetch.
	ISNP_HIT_N1	Snoop hit in cache at distance 1, instruction access.
	ISNP_HIT_N1_HWPRF	Snoop hit in cache at distance 1, hardware instruction prefetch.

Number	Mnemonic	Description
	ISNP_HIT_N1_PRF	Snoop hit in cache at distance 1, instruction prefetch.
	ISNP_HIT_N1_PRFM	Snoop hit in cache at distance 1, software instruction prefetch.
	ISNP_HIT_N1_RD	Snoop hit in cache at distance 1, demand instruction fetch.
	ISNP_HIT_N2	Snoop hit in cache at distance 2, instruction access.
	ISNP_HIT_N2_HWPRF	Snoop hit in cache at distance 2, hardware instruction prefetch.
	ISNP_HIT_N2_PRF	Snoop hit in cache at distance 2, instruction prefetch.
	ISNP_HIT_N2_PRFM	Snoop hit in cache at distance 2, software instruction prefetch.
	ISNP_HIT_N2_RD	Snoop hit in cache at distance 2, demand instruction fetch.
	ISNP_HIT_N3	Snoop hit in cache at distance 3, instruction access.
	ISNP_HIT_N3_HWPRF	Snoop hit in cache at distance 3, hardware instruction prefetch.
	ISNP_HIT_N3_PRF	Snoop hit in cache at distance 3, instruction prefetch.
	ISNP_HIT_N3_PRFM	Snoop hit in cache at distance 3, software instruction prefetch.
	ISNP_HIT_N3_RD	Snoop hit in cache at distance 3, demand instruction fetch.
	ISNP_HIT_N4	Snoop hit in cache at distance 4, instruction access.
	ISNP_HIT_N4_HWPRF	Snoop hit in cache at distance 4, hardware instruction prefetch.
	ISNP_HIT_N4_PRF	Snoop hit in cache at distance 4, instruction prefetch.
	ISNP_HIT_N4_PRFM	Snoop hit in cache at distance 4, software instruction prefetch.
	ISNP_HIT_N4_RD	Snoop hit in cache at distance 4, demand instruction fetch.
	ISNP_HIT_NEAR	Snoop hit in near local cache, instruction access.
	ISNP_HIT_NEAR_HWPRF	Snoop hit in near local cache, hardware instruction prefetch.
	ISNP_HIT_NEAR_PRF	Snoop hit in near local cache, instruction prefetch.
	ISNP_HIT_NEAR_PRFM	Snoop hit in near local cache, software instruction prefetch.
	ISNP_HIT_NEAR_RD	Snoop hit in near local cache, demand instruction fetch.
	ISNP_HIT_NEAR_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in near cache.

Number	Mnemonic	Description
	ISNP_HIT_PRF	Snoop hit, instruction prefetch.
	ISNP_HIT_PRFM	Snoop hit, software instruction prefetch.
	ISNP_HIT_RD	Snoop hit, demand instruction fetch.
	ISNP_HIT_REMOTE	Snoop hit in remote cache, instruction access.
	ISNP_HIT_REMOTE_HWPRF	Snoop hit in remote cache, hardware instruction prefetch.
	ISNP_HIT_REMOTE_PRF	Snoop hit in remote cache, instruction prefetch.
	ISNP_HIT_REMOTE_PRFM	Snoop hit in remote cache, software instruction prefetch.
	ISNP_HIT_REMOTE_RD	Snoop hit in remote cache, demand instruction fetch.
	ISNP_HIT_REMOTE_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in remote cache.
	ISNP_HIT_RETIRED	Instruction architecturally executed, instruction fetch snoop hit.
	ITLB_HIT_RETIRED	Instruction architecturally executed, no translation table walk.
	ITLB_WALK_HWPRF	Instruction TLB hardware prefetch, with at least one translation table walk.
	ITLB_WALK_PERCYC	Instruction translation table walks in progress.
	ITLB_WALK_PRF	Instruction TLB prefetch, with at least one translation table walk.
	ITLB_WALK_PRFM	Instruction TLB software prefetch, with at least one translation table walk.
	ITLB_WALK_RD	Instruction TLB demand access, with at least one translation table walk.
	ITLB_WALK_RETIRED	Instruction architecturally executed, at least one translation table walk.
	L1D_CACHE_HIT	Level 1 data cache hit.
	L1D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 1 data cache.
	L1D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 1 data cache.
	L1D_CACHE_HITM_RD	Level 1 data cache demand access hit modified, read.
	L1D_CACHE_HITM_RW	Level 1 data cache demand access hit modified.
	L1D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 1 data cache.

Number	Mnemonic	Description
	L1D_CACHE_HITM_WR	Level 1 data cache demand access hit modified, write.
	L1D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 1 data cache.
	L1D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 1 data cache.
	L1D_CACHE_HIT_PRFM	Level 1 data cache software prefetch hit.
	L1D_CACHE_HIT_RD	Level 1 data cache demand access hit, read.
	L1D_CACHE_HIT_RD_FHWPRF	Level 1 data cache demand access first hit, read, fetched by hardware prefetcher.
	L1D_CACHE_HIT_RD_FPRF	Level 1 data cache demand access first hit, read, fetched by prefetch.
	L1D_CACHE_HIT_RD_FPRFM	Level 1 data cache demand access first hit, read, fetched by software prefetch.
	L1D_CACHE_HIT_RW	Level 1 data cache demand access hit.
	L1D_CACHE_HIT_RW_FHWPRF	Level 1 data cache demand access first hit, fetched by hardware prefetcher.
	L1D_CACHE_HIT_RW_FPRF	Level 1 data cache demand access first hit, fetched by prefetch.
	L1D_CACHE_HIT_RW_FPRFM	Level 1 data cache demand access first hit, fetched by software prefetch.
	L1D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 1 data cache.
	L1D_CACHE_HIT_WR	Level 1 data cache demand access hit, write.
	L1D_CACHE_HIT_WR_FHWPRF	Level 1 data cache demand access first hit, write, fetched by hardware prefetcher.
	L1D_CACHE_HIT_WR_FPRF	Level 1 data cache demand access first hit, write, fetched by prefetch.
	L1D_CACHE_HIT_WR_FPRFM	Level 1 data cache demand access first hit, write, fetched by software prefetch.
	L1D_CACHE_HWPRF	Level 1 data cache hardware prefetch.
	L1D_CACHE_MISS	Level 1 data cache demand access miss.
	L1D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 1 data cache.
	L1D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 1 data cache.
	L1D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 1 data cache.

Number	Mnemonic	Description
	L1D_CACHE_PRF	Level 1 data cache, prefetch hit.
	L1D_CACHE_PRFM	Level 1 data cache software prefetch.
	L1D_CACHE_REFILL_HWPRF	Level 1 data cache refill, hardware prefetch.
	L1D_CACHE_REFILL_PERCYC	Level 1 data or unified cache refills in progress.
	L1D_CACHE_REFILL_PRF	Level 1 data cache refill, prefetch hit.
	L1D_CACHE_REFILL_PRFM	Level 1 data cache refill, software prefetch.
	L1D_CACHE_RW	Level 1 data cache demand access.
	L1D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
	L1D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
	L1D_LFB_HIT_RD	Level 1 data cache demand access line-fill buffer hit, read.
	L1D_LFB_HIT_RD_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
	L1D_LFB_HIT_RD_FPRF	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
	L1D_LFB_HIT_RD_FPRFM	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
	L1D_LFB_HIT_RW	Level 1 data cache demand access line-fill buffer hit.
	L1D_LFB_HIT_RW_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
	L1D_LFB_HIT_RW_FPRF	Level 1 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
	L1D_LFB_HIT_RW_FPRFM	Level 1 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
	L1D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
	L1D_LFB_HIT_WR	Level 1 data cache demand access line-fill buffer hit, write.
	L1D_LFB_HIT_WR_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
	L1D_LFB_HIT_WR_FPRF	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.

Number	Mnemonic	Description
	L1D_LFB_HIT_WR_FPRFM	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
	L1D_TLB_HWPRF	Level 1 data TLB access, hardware prefetch.
	L1D_TLB_PRF	Level 1 data TLB access, prefetch.
	L1D_TLB_PRFM	Level 1 data TLB access, software prefetch.
	L1D_TLB_RW	Level 1 data TLB access, demand access.
	L1GCS_CACHE	Level 1 GCS cache access.
	L1GCS_CACHE_HIT_RW	Level 1 GCS cache demand access hit.
	L1GCS_CACHE_HIT_RW_FHWPRF	Level 1 GCS cache demand access first hit, fetched by GCS hardware prefetcher.
	L1GCS_CACHE_HWPRF	Level 1 GCS cache hardware prefetch.
	L1GCS_CACHE_INVALID	Level 1 GCS cache invalidate.
	L1GCS_CACHE_MISS	Level 1 GCS cache demand access miss.
	L1GCS_CACHE_MISS_RD	Level 1 GCS cache demand access read miss.
	L1GCS_CACHE_REFILL	Level 1 GCS cache refill.
	L1GCS_CACHE_REFILL_HWPRF	Level 1 GCS cache refill, hardware prefetch.
	L1GCS_CACHE_REFILL_PERCYC	Level 1 GCS cache refills in progress.
	L1GCS_CACHE_RW	Level 1 GCS cache demand access.
	L1GCS_LFB_HIT_RW	Level 1 GCS cache demand access line-fill buffer hit.
	L1GCS_LFB_HIT_RW_FHWPRF	Level 1 GCS cache demand access line-fill buffer first hit, recently fetched by GCS hardware prefetcher.
	L1GCS_TLB	Level 1 GCS TLB access.
	L1GCS_TLB_HWPRF	Level 1 GCS demand TLB access, GCS hardware prefetch.
	L1GCS_TLB_RW	Level 1 GCS demand TLB access.
	L1I_CACHE_HIT	Level 1 instruction cache hit.
	L1I_CACHE_HIT_PRFM	Level 1 instruction cache software prefetch hit.
	L1I_CACHE_HIT_RD	Level 1 instruction cache demand fetch hit.
	L1I_CACHE_HIT_RD_FHWPRF	Level 1 instruction cache demand fetch first hit, fetched by hardware prefetcher.
	L1I_CACHE_HIT_RD_FPRF	Level 1 instruction cache demand fetch first hit, fetched by prefetch.

Number	Mnemonic	Description
	L1I_CACHE_HIT_RD_FPRFM	Level 1 instruction cache demand fetch first hit, fetched by software prefetch.
	L1I_CACHE_HIT_RETIRED	Instruction architecturally executed, hit in Level 1 instruction cache.
	L1I_CACHE_HWPRF	Level 1 instruction cache hardware prefetch.
	L1I_CACHE_LMISS	Level 1 instruction cache long-latency miss.
	L1I_CACHE_MISS_RETIRED	Instruction architecturally executed, miss in Level 1 instruction cache.
	L1I_CACHE_PRF	Level 1 instruction cache, prefetch hit.
	L1I_CACHE_PRFM	Level 1 instruction cache software prefetch.
	L1I_CACHE_RD	Level 1 instruction cache demand fetch.
	L1I_CACHE_REFILL_HWPRF	Level 1 instruction cache refill, hardware prefetch.
	L1I_CACHE_REFILL_PERCYC	Level 1 instruction or unified cache refills in progress.
	L1I_CACHE_REFILL_PRF	Level 1 instruction cache refill, prefetch hit.
	L1I_CACHE_REFILL_PRFM	Level 1 instruction cache refill, software prefetch.
	L1I_LFB_HIT_RD	Level 1 instruction cache demand fetch line-fill buffer hit.
	L1I_LFB_HIT_RD_FHWPRF	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher.
	L1I_LFB_HIT_RD_FPRF	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch.
	L1I_LFB_HIT_RD_FPRFM	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch.
	L1I_LFB_HIT_RETIRED	Instruction architecturally executed, line-fill buffer hit in Level 1 instruction cache.
	L1I_TLB_HWPRF	Level 1 instruction TLB access, hardware prefetch.
	L1I_TLB_PRF	Level 1 instruction TLB access, prefetch.
	L1I_TLB_PRFM	Level 1 instruction TLB access, software prefetch.
	L1I_TLB_RD	Level 1 instruction TLB access, demand access.
	L2D_CACHE_HIT	Level 2 data cache hit.

Number	Mnemonic	Description
	L2D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 2 data cache.
	L2D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 2 data cache.
	L2D_CACHE_HITM_RD	Level 2 data cache demand access hit modified, read.
	L2D_CACHE_HITM_RW	Level 2 data cache demand access hit modified.
	L2D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 2 data cache.
	L2D_CACHE_HITM_WR	Level 2 data cache demand access hit modified, write.
	L2D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 2 data cache.
	L2D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 2 data cache.
	L2D_CACHE_HIT_PRFM	Level 2 data cache software prefetch hit.
	L2D_CACHE_HIT_RD	Level 2 data cache demand access hit, read.
	L2D_CACHE_HIT_RD_FHWPRF	Level 2 data cache demand access first hit, read, fetched by hardware prefetcher.
	L2D_CACHE_HIT_RD_FPRF	Level 2 data cache demand access first hit, read, fetched by prefetch.
	L2D_CACHE_HIT_RD_FPRFM	Level 2 data cache demand access first hit, read, fetched by software prefetch.
	L2D_CACHE_HIT_RW	Level 2 data cache demand access hit.
	L2D_CACHE_HIT_RW_FHWPRF	Level 2 data cache demand access first hit, fetched by hardware prefetcher.
	L2D_CACHE_HIT_RW_FPRF	Level 2 data cache demand access first hit, fetched by prefetch.
	L2D_CACHE_HIT_RW_FPRFM	Level 2 data cache demand access first hit, fetched by software prefetch.
	L2D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 2 data cache.
	L2D_CACHE_HIT_WR	Level 2 data cache demand access hit, write.
	L2D_CACHE_HIT_WR_FHWPRF	Level 2 data cache demand access first hit, write, fetched by hardware prefetcher.
	L2D_CACHE_HIT_WR_FPRF	Level 2 data cache demand access first hit, write, fetched by prefetch.

Number	Mnemonic	Description
	L2D_CACHE_HIT_WR_FPRFM	Level 2 data cache demand access first hit, write, fetched by software prefetch.
	L2D_CACHE_HWPRF	Level 2 data cache hardware prefetch.
	L2D_CACHE_LMISS_RD	Level 2 data cache long-latency read miss.
	L2D_CACHE_MISS	Level 2 data cache demand access miss.
	L2D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 2 data cache.
	L2D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 2 data cache.
	L2D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 2 data cache.
	L2D_CACHE_PRF	Level 2 data cache, prefetch hit.
	L2D_CACHE_PRFM	Level 2 data cache software prefetch.
	L2D_CACHE_REFILL_HWPRF	Level 2 data cache refill, hardware prefetch.
	L2D_CACHE_REFILL_PERCYC	Level 2 data or unified cache refills in progress.
	L2D_CACHE_REFILL_PRF	Level 2 data cache refill, prefetch hit.
	L2D_CACHE_REFILL_PRFM	Level 2 data cache refill, software prefetch.
	L2D_CACHE_RW	Level 2 data cache demand access.
	L2D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
	L2D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
	L2D_LFB_HIT_RD	Level 2 data cache demand access line-fill buffer hit, read.
	L2D_LFB_HIT_RD_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
	L2D_LFB_HIT_RD_FPRF	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
	L2D_LFB_HIT_RD_FPRFM	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
	L2D_LFB_HIT_RW	Level 2 data cache demand access line-fill buffer hit.
	L2D_LFB_HIT_RW_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.

Number	Mnemonic	Description
	L2D_LFB_HIT_RW_FPRF	Level 2 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
	L2D_LFB_HIT_RW_FPRFM	Level 2 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
	L2D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
	L2D_LFB_HIT_WR	Level 2 data cache demand access line-fill buffer hit, write.
	L2D_LFB_HIT_WR_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
	L2D_LFB_HIT_WR_FPRF	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
	L2D_LFB_HIT_WR_FPRFM	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
	L2I_CACHE_HIT	Level 2 instruction cache hit.
	L2I_CACHE_HIT_PRFM	Level 2 instruction cache software prefetch hit.
	L2I_CACHE_HIT_RD	Level 2 instruction cache demand fetch hit.
	L2I_CACHE_HIT_RD_FHWPRF	Level 2 instruction cache demand fetch first hit, fetched by hardware prefetcher.
	L2I_CACHE_HIT_RD_FPRF	Level 2 instruction cache demand fetch first hit, fetched by prefetch.
	L2I_CACHE_HIT_RD_FPRFM	Level 2 instruction cache demand fetch first hit, fetched by software prefetch.
	L2I_CACHE_HIT_RETIRED	Instruction architecturally executed, hit in Level 2 instruction cache.
	L2I_CACHE_HWPRF	Level 2 instruction cache hardware prefetch.
	L2I_CACHE_LMISS	Level 2 instruction cache long-latency miss.
	L2I_CACHE_MISS_RETIRED	Instruction architecturally executed, miss in Level 2 instruction cache.
	L2I_CACHE_PRF	Level 2 instruction cache, prefetch hit.
	L2I_CACHE_PRFM	Level 2 instruction cache software prefetch.
	L2I_CACHE_RD	Level 2 instruction cache demand fetch.
	L2I_CACHE_REFILL_HWPRF	Level 2 instruction cache refill, hardware prefetch.
	L2I_CACHE_REFILL_PERCYC	Level 2 instruction or unified cache refills in progress.

Number	Mnemonic	Description
	L2I_CACHE_REFILL_PRF	Level 2 instruction cache refill, prefetch hit.
	L2I_CACHE_REFILL_PRFM	Level 2 instruction cache refill, software prefetch.
	L2I_LFB_HIT_RD	Level 2 instruction cache demand fetch line-fill buffer hit.
	L2I_LFB_HIT_RD_FHWPRF	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher.
	L2I_LFB_HIT_RD_FPRF	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch.
	L2I_LFB_HIT_RD_FPRFM	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch.
	L2I_LFB_HIT_RETIRED	Instruction architecturally executed, line-fill buffer hit in Level 2 instruction cache.
	L3D_CACHE_HIT	Level 3 data cache hit.
	L3D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 3 data cache.
	L3D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 3 data cache.
	L3D_CACHE_HITM_RD	Level 3 data cache demand access hit modified, read.
	L3D_CACHE_HITM_RW	Level 3 data cache demand access hit modified.
	L3D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 3 data cache.
	L3D_CACHE_HITM_WR	Level 3 data cache demand access hit modified, write.
	L3D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 3 data cache.
	L3D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 3 data cache.
	L3D_CACHE_HIT_PRFM	Level 3 data cache software prefetch hit.
	L3D_CACHE_HIT_RD	Level 3 data cache demand access hit, read.
	L3D_CACHE_HIT_RD_FHWPRF	Level 3 data cache demand access first hit, read, fetched by hardware prefetcher.
	L3D_CACHE_HIT_RD_FPRF	Level 3 data cache demand access first hit, read, fetched by prefetch.

Number	Mnemonic	Description
	L3D_CACHE_HIT_RD_FPRFM	Level 3 data cache demand access first hit, read, fetched by software prefetch.
	L3D_CACHE_HIT_RW	Level 3 data cache demand access hit.
	L3D_CACHE_HIT_RW_FHWPRF	Level 3 data cache demand access first hit, fetched by hardware prefetcher.
	L3D_CACHE_HIT_RW_FPRF	Level 3 data cache demand access first hit, fetched by prefetch.
	L3D_CACHE_HIT_RW_FPRFM	Level 3 data cache demand access first hit, fetched by software prefetch.
	L3D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 3 data cache.
	L3D_CACHE_HIT_WR	Level 3 data cache demand access hit, write.
	L3D_CACHE_HIT_WR_FHWPRF	Level 3 data cache demand access first hit, write, fetched by hardware prefetcher.
	L3D_CACHE_HIT_WR_FPRF	Level 3 data cache demand access first hit, write, fetched by prefetch.
	L3D_CACHE_HIT_WR_FPRFM	Level 3 data cache demand access first hit, write, fetched by software prefetch.
	L3D_CACHE_HWPRF	Level 3 data cache hardware prefetch.
	L3D_CACHE_LMISS_RD	Level 3 data cache long-latency read miss.
	L3D_CACHE_MISS	Level 3 data cache demand access miss.
	L3D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 3 data cache.
	L3D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 3 data cache.
	L3D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 3 data cache.
	L3D_CACHE_PRF	Level 3 data cache, prefetch hit.
	L3D_CACHE_PRFM	Level 3 data cache software prefetch.
	L3D_CACHE_REFILL_HWPRF	Level 3 data cache refill, hardware prefetch.
	L3D_CACHE_REFILL_PERCYC	Level 3 data or unified cache refills in progress.
	L3D_CACHE_REFILL_PRF	Level 3 data cache refill, prefetch hit.
	L3D_CACHE_REFILL_PRFM	Level 3 data cache refill, software prefetch.
	L3D_CACHE_RW	Level 3 data cache demand access.
	L3D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 3 data cache.

Number	Mnemonic	Description
	L3D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 3 data cache.
	L3D_LFB_HIT_RD	Level 3 data cache demand access line-fill buffer hit, read.
	L3D_LFB_HIT_RD_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
	L3D_LFB_HIT_RD_FPRF	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
	L3D_LFB_HIT_RD_FPRFM	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
	L3D_LFB_HIT_RW	Level 3 data cache demand access line-fill buffer hit.
	L3D_LFB_HIT_RW_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
	L3D_LFB_HIT_RW_FPRF	Level 3 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
	L3D_LFB_HIT_RW_FPRFM	Level 3 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
	L3D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 3 data cache.
	L3D_LFB_HIT_WR	Level 3 data cache demand access line-fill buffer hit, write.
	L3D_LFB_HIT_WR_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
	L3D_LFB_HIT_WR_FPRF	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
	L3D_LFB_HIT_WR_FPRFM	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
	LDST_ALIGN_LAT	Access with additional latency from alignment.
	LDST_ANY_RETIRED	Instruction architecturally executed, load or store.
	LDST_FIXED_BYTES_SPEC	Non-scalable load and store bytes speculatively executed.
	LDST_FIXED_OPS_SPEC	Non-scalable load or store element operations speculatively executed.
	LD_ALIGN_LAT	Load with additional latency from alignment.
	LD_ANY_RETIRED	Instruction architecturally executed, load.

Number	Mnemonic	Description
	LD_FIXED_BYTES_SPEC	Non-scalable load bytes speculatively executed.
	LD_FIXED_OPS_SPEC	Non-scalable load element operations speculatively executed.
	LL_CACHE_HIT	Last level cache hit.
	LL_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Last level cache.
	LL_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Last level cache.
	LL_CACHE_HITM_RD	Last level cache demand access hit modified, read.
	LL_CACHE_HITM_RW	Last level cache demand access hit modified.
	LL_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Last level cache.
	LL_CACHE_HITM_WR	Last level cache demand access hit modified, write.
	LL_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Last level cache.
	LL_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Last level cache.
	LL_CACHE_HIT_PRFM	Last level cache software prefetch hit.
	LL_CACHE_HIT_RD	Last level cache demand access hit, read.
	LL_CACHE_HIT_RD_FHWPRF	Last level cache demand access first hit, read, fetched by hardware prefetcher.
	LL_CACHE_HIT_RD_FPRF	Last level cache demand access first hit, read, fetched by prefetch.
	LL_CACHE_HIT_RD_FPRFM	Last level cache demand access first hit, read, fetched by software prefetch.
	LL_CACHE_HIT_RW	Last level cache demand access hit.
	LL_CACHE_HIT_RW_FHWPRF	Last level cache demand access first hit, fetched by hardware prefetcher.
	LL_CACHE_HIT_RW_FPRF	Last level cache demand access first hit, fetched by prefetch.
	LL_CACHE_HIT_RW_FPRFM	Last level cache demand access first hit, fetched by software prefetch.
	LL_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Last level cache.
	LL_CACHE_HIT_WR	Last level cache demand access hit, write.
	LL_CACHE_HIT_WR_FHWPRF	Last level cache demand access first hit, write, fetched by hardware prefetcher.

Number	Mnemonic	Description
	LL_CACHE_HIT_WR_FPRF	Last level cache demand access first hit, write, fetched by prefetch.
	LL_CACHE_HIT_WR_FPRFM	Last level cache demand access first hit, write, fetched by software prefetch.
	LL_CACHE_HWPRF	Last level cache hardware prefetch.
	LL_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Last level cache.
	LL_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Last level cache.
	LL_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Last level cache.
	LL_CACHE_PRF	Last level cache, prefetch hit.
	LL_CACHE_PRFM	Last level cache software prefetch.
	LL_CACHE_REFILL	Last level cache refill.
	LL_CACHE_REFILL_HWPRF	Last level cache refill, hardware prefetch.
	LL_CACHE_REFILL_PRF	Last level cache refill, prefetch hit.
	LL_CACHE_REFILL_PRFM	Last level cache refill, software prefetch.
	LL_CACHE_REFILL_WR	Last level cache refill, write.
	LL_CACHE_RW	Last level cache demand access.
	LL_CACHE_WB	Last level cache write-back.
	LL_CACHE_WR	Last level cache access, write.
	LL_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Last level cache.
	LL_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Last level cache.
	LL_LFB_HIT_RD	Last level cache demand access line-fill buffer hit, read.
	LL_LFB_HIT_RD_FHWPRF	Last level cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
	LL_LFB_HIT_RD_FPRF	Last level cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
	LL_LFB_HIT_RD_FPRFM	Last level cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
	LL_LFB_HIT_RW	Last level cache demand access line-fill buffer hit.
	LL_LFB_HIT_RW_FHWPRF	Last level cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.

Number	Mnemonic	Description
	LL_LFB_HIT_RW_FPRF	Last level cache demand access line-fill buffer first hit, recently fetched by prefetch.
	LL_LFB_HIT_RW_FPRFM	Last level cache demand access line-fill buffer first hit, recently fetched by software prefetch.
	LL_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Last level cache.
	LL_LFB_HIT_WR	Last level cache demand access line-fill buffer hit, write.
	LL_LFB_HIT_WR_FHWPRF	Last level cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
	LL_LFB_HIT_WR_FPRF	Last level cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
	LL_LFB_HIT_WR_FPRFM	Last level cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
	LOCAL_MEM	Access to memory attached to this device.
	LOCAL_MEM_LDST_RETIRED	Load or store instruction architecturally executed, access to memory attached to this device.
	LOCAL_MEM_LD_RETIRED	Load instruction architecturally executed, access to memory attached to this device.
	LOCAL_MEM_PRFM	Access to memory attached to this device, software prefetch.
	LOCAL_MEM_RD	Access to memory attached to this device, demand access, read.
	LOCAL_MEM_RW	Access to memory attached to this device, demand access.
	LOCAL_MEM_ST_RETIRED	Store instruction architecturally executed, access to memory attached to this device.
	LOCAL_MEM_WR	Access to memory attached to this device, demand access, write.
	MEM_ACCESS_PRFM	Data memory access, software prefetch.
	MEM_ACCESS_RD_PERCYC	Data memory reads in progress.
	MEM_ACCESS_RW	Data memory access, demand access.
	N1_CACHE1_HITM_RD	Cache type 1 at distance 1 demand access hit modified, read.
	N1_CACHE1_HIT_PRFM	Cache type 1 at distance 1 software prefetch hit.
	N1_CACHE1_HIT_RD	Cache type 1 at distance 1 demand access hit, read.

Number	Mnemonic	Description
	N1_CACHE2_HITM_RD	Cache type 2 at distance 1 demand access hit modified, read.
	N1_CACHE2_HIT_PRFM	Cache type 2 at distance 1 software prefetch hit.
	N1_CACHE2_HIT_RD	Cache type 2 at distance 1 demand access hit, read.
	N1_CACHE3_HITM_RD	Cache type 3 at distance 1 demand access hit modified, read.
	N1_CACHE3_HIT_PRFM	Cache type 3 at distance 1 software prefetch hit.
	N1_CACHE3_HIT_RD	Cache type 3 at distance 1 demand access hit, read.
	N1_CACHE_HITM_RD	Cache at distance 1, demand access hit modified, read.
	N1_CACHE_HIT_PRFM	Cache at distance 1, software prefetch hit.
	N1_CACHE_HIT_RD	Cache at distance 1, demand access hit, read.
	N1_LFB1_HIT_RD	Cache type 1 at distance 1 demand access line-fill buffer hit, read.
	N1_LFB2_HIT_RD	Cache type 2 at distance 1 demand access line-fill buffer hit, read.
	N1_LFB3_HIT_RD	Cache type 3 at distance 1 demand access line-fill buffer hit, read.
	N1_LFB_HIT_RD	Cache at distance 1 demand access line-fill buffer hit, read.
	N1_MEM1_PRFM	Access to type 1 memory at distance 1, software prefetch.
	N1_MEM1_RD	Access to type 1 memory at distance 1, demand access, read.
	N1_MEM2_PRFM	Access to type 2 memory at distance 1, software prefetch.
	N1_MEM2_RD	Access to type 2 memory at distance 1, demand access, read.
	N1_MEM3_PRFM	Access to type 3 memory at distance 1, software prefetch.
	N1_MEM3_RD	Access to type 3 memory at distance 1, demand access, read.
	N1_MEM_PRFM	Access to memory at distance 1, software prefetch.
	N1_MEM_RD	Access to memory at distance 1, demand access, read.
	N2_CACHE1_HITM_RD	Cache type 1 at distance 2 demand access hit modified, read.
	N2_CACHE1_HIT_PRFM	Cache type 1 at distance 2 software prefetch hit.
	N2_CACHE1_HIT_RD	Cache type 1 at distance 2 demand access hit, read.

Number	Mnemonic	Description
	N2_CACHE2_HITM_RD	Cache type 2 at distance 2 demand access hit modified, read.
	N2_CACHE2_HIT_PRFM	Cache type 2 at distance 2 software prefetch hit.
	N2_CACHE2_HIT_RD	Cache type 2 at distance 2 demand access hit, read.
	N2_CACHE3_HITM_RD	Cache type 3 at distance 2 demand access hit modified, read.
	N2_CACHE3_HIT_PRFM	Cache type 3 at distance 2 software prefetch hit.
	N2_CACHE3_HIT_RD	Cache type 3 at distance 2 demand access hit, read.
	N2_CACHE_HITM_RD	Cache at distance 2, demand access hit modified, read.
	N2_CACHE_HIT_PRFM	Cache at distance 2, software prefetch hit.
	N2_CACHE_HIT_RD	Cache at distance 2, demand access hit, read.
	N2_LFB1_HIT_RD	Cache type 1 at distance 2 demand access line-fill buffer hit, read.
	N2_LFB2_HIT_RD	Cache type 2 at distance 2 demand access line-fill buffer hit, read.
	N2_LFB3_HIT_RD	Cache type 3 at distance 2 demand access line-fill buffer hit, read.
	N2_LFB_HIT_RD	Cache at distance 2 demand access line-fill buffer hit, read.
	N2_MEM1_PRFM	Access to type 1 memory at distance 2, software prefetch.
	N2_MEM1_RD	Access to type 1 memory at distance 2, demand access, read.
	N2_MEM2_PRFM	Access to type 2 memory at distance 2, software prefetch.
	N2_MEM2_RD	Access to type 2 memory at distance 2, demand access, read.
	N2_MEM3_PRFM	Access to type 3 memory at distance 2, software prefetch.
	N2_MEM3_RD	Access to type 3 memory at distance 2, demand access, read.
	N2_MEM_PRFM	Access to memory at distance 2, software prefetch.
	N2_MEM_RD	Access to memory at distance 2, demand access, read.
	N3_CACHE1_HITM_RD	Cache type 1 at distance 3 demand access hit modified, read.
	N3_CACHE1_HIT_PRFM	Cache type 1 at distance 3 software prefetch hit.
	N3_CACHE1_HIT_RD	Cache type 1 at distance 3 demand access hit, read.

Number	Mnemonic	Description
	N3_CACHE2_HITM_RD	Cache type 2 at distance 3 demand access hit modified, read.
	N3_CACHE2_HIT_PRFM	Cache type 2 at distance 3 software prefetch hit.
	N3_CACHE2_HIT_RD	Cache type 2 at distance 3 demand access hit, read.
	N3_CACHE3_HITM_RD	Cache type 3 at distance 3 demand access hit modified, read.
	N3_CACHE3_HIT_PRFM	Cache type 3 at distance 3 software prefetch hit.
	N3_CACHE3_HIT_RD	Cache type 3 at distance 3 demand access hit, read.
	N3_CACHE_HITM_RD	Cache at distance 3, demand access hit modified, read.
	N3_CACHE_HIT_PRFM	Cache at distance 3, software prefetch hit.
	N3_CACHE_HIT_RD	Cache at distance 3, demand access hit, read.
	N3_LFB1_HIT_RD	Cache type 1 at distance 3 demand access line-fill buffer hit, read.
	N3_LFB2_HIT_RD	Cache type 2 at distance 3 demand access line-fill buffer hit, read.
	N3_LFB3_HIT_RD	Cache type 3 at distance 3 demand access line-fill buffer hit, read.
	N3_LFB_HIT_RD	Cache at distance 3 demand access line-fill buffer hit, read.
	N3_MEM1_PRFM	Access to type 1 memory at distance 3, software prefetch.
	N3_MEM1_RD	Access to type 1 memory at distance 3, demand access, read.
	N3_MEM2_PRFM	Access to type 2 memory at distance 3, software prefetch.
	N3_MEM2_RD	Access to type 2 memory at distance 3, demand access, read.
	N3_MEM3_PRFM	Access to type 3 memory at distance 3, software prefetch.
	N3_MEM3_RD	Access to type 3 memory at distance 3, demand access, read.
	N3_MEM_PRFM	Access to memory at distance 3, software prefetch.
	N3_MEM_RD	Access to memory at distance 3, demand access, read.
	N4_CACHE1_HITM_RD	Cache type 1 at distance 4 demand access hit modified, read.
	N4_CACHE1_HIT_PRFM	Cache type 1 at distance 4 software prefetch hit.
	N4_CACHE1_HIT_RD	Cache type 1 at distance 4 demand access hit, read.

Number	Mnemonic	Description
	N4_CACHE2_HITM_RD	Cache type 2 at distance 4 demand access hit modified, read.
	N4_CACHE2_HIT_PRFM	Cache type 2 at distance 4 software prefetch hit.
	N4_CACHE2_HIT_RD	Cache type 2 at distance 4 demand access hit, read.
	N4_CACHE3_HITM_RD	Cache type 3 at distance 4 demand access hit modified, read.
	N4_CACHE3_HIT_PRFM	Cache type 3 at distance 4 software prefetch hit.
	N4_CACHE3_HIT_RD	Cache type 3 at distance 4 demand access hit, read.
	N4_CACHE_HITM_RD	Cache at distance 4, demand access hit modified, read.
	N4_CACHE_HIT_PRFM	Cache at distance 4, software prefetch hit.
	N4_CACHE_HIT_RD	Cache at distance 4, demand access hit, read.
	N4_LFB1_HIT_RD	Cache type 1 at distance 4 demand access line-fill buffer hit, read.
	N4_LFB2_HIT_RD	Cache type 2 at distance 4 demand access line-fill buffer hit, read.
	N4_LFB3_HIT_RD	Cache type 3 at distance 4 demand access line-fill buffer hit, read.
	N4_LFB_HIT_RD	Cache at distance 4 demand access line-fill buffer hit, read.
	N4_MEM1_PRFM	Access to type 1 memory at distance 4, software prefetch.
	N4_MEM1_RD	Access to type 1 memory at distance 4, demand access, read.
	N4_MEM2_PRFM	Access to type 2 memory at distance 4, software prefetch.
	N4_MEM2_RD	Access to type 2 memory at distance 4, demand access, read.
	N4_MEM3_PRFM	Access to type 3 memory at distance 4, software prefetch.
	N4_MEM3_RD	Access to type 3 memory at distance 4, demand access, read.
	N4_MEM_PRFM	Access to memory at distance 4, software prefetch.
	N4_MEM_RD	Access to memory at distance 4, demand access, read.
	NONFP_SPEC	Non-floating-point operation speculatively executed.
	PMU_SNAPSHOT	Successful PMU capture event.
	PRF_RETIRED	Instruction architecturally executed, prefetch.

Number	Mnemonic	Description
	PRF_SPEC	Operation speculatively executed, prefetch.
	REMOTE_ACCESS_WR	Access to a remote device, write.
	REMOTE_CACHE_HITM_RD	Remote cache demand access hit modified, read.
	REMOTE_CACHE_HIT_PRFM	Remote cache software prefetch hit.
	REMOTE_CACHE_HIT_RD	Remote cache demand access hit, read.
	REMOTE_LFB_HIT_RD	Remote cache demand access line-fill buffer hit, read.
	REMOTE_MEM	Access to memory attached to a remote device.
	REMOTE_MEM_LDST_RETIRED	Load or store instruction architecturally executed, access to memory attached to a remote device.
	REMOTE_MEM_LD_RETIRED	Load instruction architecturally executed, access to memory attached to a remote device.
	REMOTE_MEM_PRFM	Access to memory attached to a remote device, software prefetch.
	REMOTE_MEM_RD	Access to memory attached to a remote device, demand access, read.
	REMOTE_MEM_RW	Access to memory attached to a remote device, demand access.
	REMOTE_MEM_ST_RETIRED	Store instruction architecturally executed, access to memory attached to a remote device.
	REMOTE_MEM_WR	Access to memory attached to a remote device, demand access, write.
	SAMPLE_BUFFER_FULL	Profiling Buffer full.
	SAMPLE_COLLISION	Statistical Profiling sample collided with previous sample.
	SAMPLE_FEED	Statistical Profiling sample taken.
	SAMPLE_FEED_DS	Statistical Profiling sample taken, selected Data Source.
	SAMPLE_FILTRATE	Statistical Profiling sample taken and not removed by filtering.
	SAMPLE_POP	Statistical Profiling sample population.
	SE_FP_ADDSUB_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point addition or subtraction.
	SE_FP_BF16_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is BFloat16 floating-point.

Number	Mnemonic	Description
	SE_FP_BF16_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD BFloat16 floating-point, data-processing.
	SE_FP_DOT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point dot product.
	SE_FP_DP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD double-precision floating-point, data-processing.
	SE_FP_FMA_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point multiply-add or multiply-subtract.
	SE_FP_FP8_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is 8-bit floating-point.
	SE_FP_HP_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is half-precision floating-point.
	SE_FP_HP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD half-precision floating-point, data-processing.
	SE_FP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point, data-processing.
	SE_FP_SP_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is single-precision floating-point.
	SE_FP_SP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD single-precision floating-point, data-processing.
	SE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD, SVE, or SME.
	SE_INST_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME.
	SE_INT16_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 16-bit integer, data-processing.
	SE_INT32_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 32-bit integer, data-processing.
	SE_INT64_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 64-bit integer, data-processing.

Number	Mnemonic	Description
	SE_INT8_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 8-bit integer, data-processing.
	SE_INT_DOT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer dot product.
	SE_INT_MUL_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer multiply or multiply-accumulate.
	SE_INT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer, data-processing.
	SE_LUT_SPEC	Lookup table operation speculatively executed, Advanced SIMD, SVE, or SME data processing.
	SE_RETIRED	Instruction architecturally executed, Advanced SIMD, SVE, or SME data processing.
	SE_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME data processing.
	SIMD_INST_RETIRED	Instruction architecturally executed, SIMD.
	SIMD_INST_SPEC	Operation speculatively executed, SIMD.
	SIMD_UOP_SPEC	Microarchitectural operation speculatively executed, SIMD.
	SME_FP_BF16_MIN_SPEC	SME data processing operation speculatively executed, smallest type is BFloat16 floating-point.
	SME_FP_FP8_MIN_SPEC	SME data processing operation speculatively executed, smallest type is 8-bit floating-point.
	SME_FP_HP_MIN_SPEC	SME data processing operation speculatively executed, smallest type is half-precision floating-point.
	SME_FP_SP_MIN_SPEC	SME data processing operation speculatively executed, smallest type is single-precision floating-point.
	STALL_BACKEND_BUSY	Backend stall cycles, backend busy.
	STALL_BACKEND_CPUBOUND	Backend stall cycles, processor bound.
	STALL_BACKEND_GCSTLB	Backend stall cycles, GCS TLB.
	STALL_BACKEND_ILOCK	Backend stall cycles, input dependency.
	STALL_BACKEND_L1D	Backend stall cycles, level 1 data cache.
	STALL_BACKEND_L1GCS	Backend stall cycles, Level 1 GCS cache.

Number	Mnemonic	Description
	STALL_BACKEND_L2D	Backend stall cycles, level 2 data cache.
	STALL_BACKEND_MEM	Memory stall cycles.
	STALL_BACKEND_MEMBOUND	Backend stall cycles, memory bound.
	STALL_BACKEND_RENAME	Backend stall cycles, rename full.
	STALL_BACKEND_ST	Backend stall cycles, store.
	STALL_BACKEND_TLB	Backend stall cycles, TLB.
	STALL_FRONTEND_CPUBOUND	Frontend stall cycles, processor bound.
	STALL_FRONTEND_FLOW	Frontend stall cycles, flow control.
	STALL_FRONTEND_FLUSH	Frontend stall cycles, flush recovery.
	STALL_FRONTEND_L1I	Frontend stall cycles, level 1 instruction cache.
	STALL_FRONTEND_L2I	Frontend stall cycles, level 2 instruction cache.
	STALL_FRONTEND_MEM	Frontend stall cycles, last level PE cache or memory.
	STALL_FRONTEND_MEMBOUND	Frontend stall cycles, memory bound.
	STALL_FRONTEND_RENAME	Frontend stall cycles, rename full.
	STALL_FRONTEND_TLB	Frontend stall cycles, TLB.
	ST_ALIGN_LAT	Store with additional latency from alignment.
	ST_ANY_RETIRED	Instruction architecturally executed, store.
	ST_FIXED_BYTES_SPEC	Non-scalable store bytes speculatively executed.
	ST_FIXED_OPS_SPEC	Non-scalable store element operations speculatively executed.
	SVE_FP_BF16_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point.
	SVE_FP_FP8_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is 8-bit floating-point.
	SVE_FP_HP_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is half-precision floating-point.
	SVE_FP_PREDUCE_SPEC	Floating-point operation speculatively executed, SVE pairwise add step or pairwise reduce step.
	SVE_FP_SP_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is single-precision floating-point.
	SVE_LUT_SPEC	Lookup table operation speculatively executed, SVE data processing.
	SVE_SPEC	Operation speculatively executed, SVE data processing.

Number	Mnemonic	Description
	TRB_TRIG	Trace buffer Trigger Event.
	TRB_WRAP	Trace buffer current write pointer wrapped.
	TRCEXTOUT0	Trace unit external output 0.
	TRCEXTOUT1	Trace unit external output 1.
	TRCEXTOUT2	Trace unit external output 2.
	TRCEXTOUT3	Trace unit external output 3.
	UOP_RETIRED	Micro-operation architecturally executed.
	UOP_SPEC	Microarchitectural operation speculatively executed.
	VFP_RETIRED	Instruction architecturally executed, scalar floating-point data processing.
	ZA_ACTIVE	CPU cycles, ZA enabled.

D7.3 Common event numbers

The event numbers of the common architectural and microarchitectural events are reserved for the specified events. Each of these event numbers must either:

- Be used for its assigned event.
- Not be used.

However, see [Required events](#).

When an implementation supports monitoring of an event that is assigned a common architectural or microarchitectural event number, Arm strongly recommends that it uses that number for the event. However, software might encounter implementations where an event assigned a number in this range is monitored using an event number from an IMPLEMENTATION DEFINED range.

Note

Arm might define other common architectural and microarchitectural event numbers. This is one reason why software must not assume that an event with an assigned common architectural or microarchitectural event number is never monitored using an event number from the IMPLEMENTATION DEFINED range.

It is IMPLEMENTATION DEFINED which events, including Common events, are generated by IMPLEMENTATION DEFINED extensions to the architecture, including accesses to IMPLEMENTATION DEFINED System registers and IMPLEMENTATION DEFINED System instructions. However, the functionality of the IMPLEMENTATION DEFINED extension must be appropriate for the generated events.

The common events are described in the following sections:

- [Common architectural events](#).
- [Common microarchitectural events](#).

The supported common architectural and microarchitectural events in the range 0x0000-0x003F are discoverable to software through the [PMCEID0](#) or [PMCEID1](#) register.

Arm recommends that the value of 0 is used for the [PMCEID0](#) or [PMCEID1](#) bit corresponding to any event that an implementation never generates, even if the implementation is considered to support but never count the event.

For example, if an implementation never generates the L1D_CACHE_ALLOCATE event, event 31, Arm recommends that [PMCEID0](#) [31] is RAZ.

However, for some implementations, an event in the common events range might be generated by the system, meaning behavior can vary between systems. In such a case, the corresponding [PMCEID \$n\$](#) bit might be RAO.

Event numbers that [Table D7-2](#) shows as allocated for common architectural and microarchitectural events that are not described in [Common architectural events](#) and [Common microarchitectural events](#) are reserved. Future revisions of this Manual, or of the architecture, might assign these reserved values to additional common events. Events that do not require additional features in the PMU can be implemented retrospectively, meaning an implementation of a particular version of the PMU specification might support common events that are first defined in a later version of the PMU specification.

Note

The requirement that an event that is implemented retrospectively *does not require additional features in the PMU* means that it must be possible to represent the event n in the [PMEVTYPER< \$n\$ >.evtCount](#) field. This means, for example, that an implementation with a 10-bit [PMEVTYPER< \$n\$ >.evtCount](#) field can only implement events with event numbers 0x0000-0x03FF.

D7.3.1 Common architectural events

The Common architectural events are:

0x0000, **SW_INCR**, Instruction architecturally executed, Condition code check pass, software increment

The counter counts each write to the PMSWINC_EL0 and **PMSWINC** register, for each implemented event counter <n>:

- In AArch64 state, if **PMEVTYPER**<n>_EL0.evtCount is 0x0000 then the counter counts each MSR write to PMSWINC_EL0 with bit [n] set to 1.
- In AArch32 state, if **PMEVTYPER**<n>_EL0.evtCount is 0x0000 then the counter counts each MCR write to **PMSWINC** with bit [n] set to 1.

If the PE performs two architecturally executed writes to the PMSWINC_EL0 or **PMSWINC** register without an intervening **Context synchronization event**, then the counter is incremented twice.

Note

In a multithreaded implementation, if the Effective value of **PMEVTYPER**<n>_EL0.MT is 1, then the counter counts writes by all PEs that have the same affinity at level 1 and above.

PMCEID0_EL0[0] reads as 1 if this event is implemented and 0 otherwise.

0x0006, **LD_RETIRED**, Instruction architecturally executed, Condition code check pass, load

The counter counts each architecturally-executed memory-reading instruction counted by **INST_RETIRED**.

It is IMPLEMENTATION DEFINED whether the prefetch instructions counted by **PRF_RETIRED** are counted as Memory-reading instructions:

- If the prefetch instructions are counted as Memory-reading instructions, then they are counted by **LD_RETIRED** and **LD_ANY_RETIRED**.
- Otherwise, if the prefetch instructions are not counted as Memory-reading instructions by **LD_RETIRED**, then it is further IMPLEMENTATION DEFINED which one of the following applies:
 - They are counted as data-processing instructions.
 - They are counted by **LDST_ANY_RETIRED**.

Arm recommends that if a prefetch instruction is not implemented as a NOP and the **PRF_RETIRED** event is not implemented, then the instruction is counted as a Memory-reading instruction.

PMCEID0_EL0[6] reads as 1 if this event is implemented and 0 otherwise.

0x0007, **ST_RETIRED**, Instruction architecturally executed, Condition code check pass, store

The counter counts each architecturally-executed **memory-writing instruction** counted by **INST_RETIRED**.

The counter does not count a Store-Exclusive instruction that fails.

PMCEID0_EL0[7] reads as 1 if this event is implemented and 0 otherwise.

0x0008, **INST_RETIRED**, Instruction architecturally executed

The counter counts each architecturally-executed instruction.

PMCEID0_EL0[8] reads as 1 if this event is implemented and 0 otherwise.

0x0009, **EXC_TAKEN**, Exception taken

The counter counts each exception taken.

It is IMPLEMENTATION SPECIFIC whether the counter counts each entry to Debug state.

Note

The counter counts the PE exceptions described in:

- For exceptions taken to an Exception level using AArch64, Exception entry.
- For exceptions taken to an Exception level using AArch32, [AArch32 state exception descriptions](#).

PMCEID0_EL0[9] reads as 1 if this event is implemented and 0 otherwise.

0x000A, **EXC_RETURN**, Instruction architecturally executed, Condition code check pass, exception return

The counter counts each architecturally-executed exception return instruction.

Instructions defined in the following sections are counted as exception return instructions:

- For an exception return from an Exception level using AArch64, Exception return.
- For an exception return from an Exception level using AArch32, [Exception return instructions](#).

It is IMPLEMENTATION SPECIFIC whether the execution of an exception return instruction is counted if any of the following apply:

- Execution of the instruction is, itself, CONSTRAINED UNPREDICTABLE.
- Execution of an exception return instruction that generates an exception.
- Execution of the instruction sets PSTATE.IL and does not generate an exception return.
- Exit from Debug state.

Note

- Examples of when an exception return instruction is CONSTRAINED UNPREDICTABLE are if the instruction is executed in AArch32 state at EL0 or in System mode.
- A particular consequence of this CONSTRAINED UNPREDICTABLE behavior is that an implementation that does not support AArch32 state at EL1 or higher does not have to count AArch32 `MOVS PC, LR`, and related instructions, as exception return instructions.

PMCEID0_EL0[10] reads as 1 if this event is implemented and 0 otherwise.

0x000B, **CID_WRITE_RETIRED**, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR

The counter counts each MSR write to CONTEXTIDR_EL1 and MCR write to [CONTEXTIDR](#).

If the PE performs two architecturally-executed writes to CONTEXTIDR without an intervening [Context synchronization event](#), it is CONSTRAINED UNPREDICTABLE whether the first write is counted.

PMCEID0_EL0[11] reads as 1 if this event is implemented and 0 otherwise.

0x000C, **PC_WRITE_RETIRED**, Instruction architecturally executed, Condition code check pass, Software change of the PC

The counter counts each architecturally-executed [Software change of the PC](#) that is taken.

This includes all:

- Branch instructions.
- ‘Memory-reading instructions’ that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.
- Exception return instructions.

Note

Conditional branches are only counted if the branch is taken.

It is IMPLEMENTATION DEFINED whether the following instructions are included as a [Software change of the PC](#):

- A BRK or BKPT instruction.
- An UNDEFINED instruction that generates an exception.
- An exception-generating instruction, SVC, HVC, and SMC, that generates an exception.
- A Context synchronization barrier instruction, ISB.

The counter does not increment for exceptions other than those explicitly identified in these lists.

PMCEID0_EL0[12] reads as 1 if this event is implemented and 0 otherwise.

0x000D, **BR_IMMED_RETIRED**, Branch instruction architecturally executed, immediate

The counter counts each architecturally-executed immediate branch instruction.

The following instructions are counted as immediate branch instructions:

- For AArch32 state, the following instructions:
 - B{<c>} <label>.
 - BL{<c>} <label>.
 - BLX{<c>} <label>.
 - CBZ <Rn>, <label>.
 - CBNZ <label>.
- For AArch64 state, the following instructions:
 - B <label>.
 - B.cond <label>.
 - BL <label>.
 - CBZ <Rn>, <label>.
 - CBNZ <Rn>, <label>.
 - TBZ <Rn>, <label>.
 - TBNZ <Rn>, <label>.

Note

Conditional branches are always counted, regardless of whether the branch is taken or not taken.

If the Context synchronization barrier instruction ISB is counted as a [Software change of the PC](#) instruction by [PC_WRITE_RETIRED](#), then it is IMPLEMENTATION DEFINED whether ISB is counted as an immediate branch instruction.

PMCEID0_EL0[13] reads as 1 if this event is implemented and 0 otherwise.

0x000E, **BR_RETURN_RETIRED**, Branch instruction architecturally executed, procedure return, taken

The counter counts each [architecturally executed](#) procedure return instruction.

The following instructions are counted as procedure return instructions:

- For AArch32 state, the following instructions:
 - BX R14.
 - MOV PC, LR.
 - POP {..., PC}.
 - LDR PC, [SP], #offset.
- For AArch64 state, the following instructions:
 - RET.

Note

The counter counts only the listed instructions as procedure returns. For example, it does not count the following AArch32 instructions as procedure return instructions:

- BX R0, because Rm != R14.
 - MOV PC, R0, because Rm != R14.
 - LDM SP, {..., PC}, because writeback is not specified.
 - LDR PC, [SP, #offset], because this specifies the wrong addressing mode.
-

PMCEID0_EL0[14] reads as 1 if this event is implemented and 0 otherwise.

0x000F, **UNALIGNED_LDST_RETIRED**, Instruction architecturally executed, Condition code check pass, unaligned load or store

The counter counts each memory-reading instruction or [memory-writing instruction](#) access that would generate an Alignment fault when Alignment fault checking is enabled.

The counter does not count accesses that would generate an SP alignment fault exception if the applicable stack pointer alignment check is enabled, unless that access would also generate an Alignment fault Data Abort exception if Alignment fault checking is enabled.

It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this event counts accesses that generate an exception, including accesses that do generate Alignment fault Data Abort exceptions.

PMCEID0_EL0[15] reads as 1 if this event is implemented and 0 otherwise.

0x001C, **TTBR_WRITE_RETIRED**, Instruction architecturally executed, Condition code check pass, write to TTBR

The counter counts each MSR write to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and MCR and MCRR write to TTBR0 and TTBR1 in AArch32 state.

The counter counts MSR writes to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and MCR and MCRR writes to TTBR0 and TTBR1 in AArch32 state.

If the PE executes two writes to the same TTBR, without an intervening [Context synchronization event](#), it is CONSTRAINED UNPREDICTABLE whether the first write to the TTBR, is counted.

The counter counts only writes to these registers by name. For example:

- If EL3 is implemented, the counter does not count writes using the register name TTBR0_EL3. However, when EL3 is using AArch32, the counter counts writes to either banked copy of TTBR0 and TTBR1.
- If EL2 is implemented, the counter does not count writes using the register names TTBR0_EL2, VTTBR_EL2, HTTBR, and VTTBR.

PMCEID0_EL0[28] reads as 1 if this event is implemented and 0 otherwise.

0x001E, **CHAIN**, Chain a pair of event counters

Even-numbered counters never increment as a result of this event.

For an odd-numbered counter <n+1>, the counter increments when an event increments the preceding even-numbered counter <n> on the same PE causing unsigned overflow of bits [31:0] of the event counter <n>.

This means the [CHAIN](#) event can be used to link the odd-numbered counter with the preceding even-numbered counter to provide a 64-bit counter.

If EL2 is implemented and <n+1> is equal to the Effective value of HDCR.HPMN, then it is UNPREDICTABLE whether the counter counts.

Note

- The **CHAIN** event allows software to use the N event counters as N 32-bit counters, N÷2 64-bit counters, or a mixture of 32-bit counters and 64-bit counters.
 - The **CHAIN** event only counts overflows from the preceding even-numbered counter on the same PE. This means it ignores the Effective value of PMEVTYPEPER<n>_EL0.MT.
 - The architecture does not provide atomic access to a pair of counters.
-

To filter the Exception levels and Security states in which the event is counted, software:

- Programs PMEVTYPEPER<n>_EL0 to count the event in the required conditions.
- Programs PMEVTYPEPER<n+1>_EL0 to count the **CHAIN** event in all Exception levels and states.

The PE might ignore the filter settings for the **CHAIN** event and behave as if they are set to count in all Exception levels and states. If software does not program the event in this way, the count becomes UNPREDICTABLE.

The architecture does not define the latency between the low counter overflowing and the high counter incrementing the **CHAIN** event. There is no requirement for updates to occur synchronously, but software reading or enabling the counter pair by reading the low counter first and the high counter second, with an intervening Context synchronization event, will not observe the low counter incrementing and overflowing for the event and the high counter not incrementing for the resulting **CHAIN** event. This means that the ISB executed after reading the low counter ensures the completion of the update of the high counter by the **CHAIN** event.

PMCEID0_EL0[30] reads as 1 if this event is implemented and 0 otherwise.

0x0021, **BR_RETIRED**, Instruction architecturally executed, branch

The counter counts each branch instruction on the architecturally executed path that would incur cost if mispredicted.

It is IMPLEMENTATION DEFINED whether the counter increments for any or all of:

- Unconditional direct branch instructions.
- Exception-generating instructions.
- Exception return instructions.
- Context synchronization instructions.

The counter counts all other branch instructions, memory-reading instructions and data-processing instructions that explicitly write to the PC, at retirement.

Note

Conditional branches are always counted, regardless of whether the branch is taken.

Arm recommends that **BR_RETIRED** counts Unconditional direct branch instructions and Exception return instructions.

PMCEID1_EL0[1] reads as 1 if this event is implemented and 0 otherwise.

0x0081, **EXC_UNDEF**, Exception taken, other synchronous

The counter counts each exception counted by **EXC_TAKEN** that is **Taken locally** and is not counted as any of the following:

- Exception taken, Supervisor Call (EXC_SVC).
- Exception taken, Secure Monitor Call (EXC_SMC).
- Exception taken, Hypervisor Call (EXC_HVC).
- Exception taken, IRQ (EXC_IRQ).
- Exception taken, FIQ (EXC_FIQ).

It is IMPLEMENTATION SPECIFIC whether the counter counts each entry to Debug state not counted by either of the following:

- Exception taken, Instruction Abort (EXC_PABORT).
- Exception taken, Data Abort or SError (EXC_DABORT).

0x0082, **EXC_SVC, Exception taken, Supervisor Call**

The counter counts each Supervisor Call exception counted by [EXC_TAKEN](#) that is [Taken locally](#).

0x0083, **EXC_PABORT, Exception taken, Instruction Abort**

The counter counts each Instruction Abort exception counted by [EXC_TAKEN](#) that is [Taken locally](#).

It is IMPLEMENTATION SPECIFIC whether the counter counts each entry to Debug state due to a Breakpoint debug event.

0x0084, **EXC_DABORT, Exception taken, Data Abort or SError**

The counter counts each Data Abort, SError interrupt, or virtual SError interrupt exception counted by [EXC_TAKEN](#) that is [Taken locally](#).

It is IMPLEMENTATION SPECIFIC whether the counter counts each entry to Debug state due to a Watchpoint debug event.

0x0086, **EXC_IRQ, Exception taken, IRQ**

The counter counts each IRQ or virtual IRQ exception counted by [EXC_TAKEN](#) that is [Taken locally](#).

0x0087, **EXC_FIQ, Exception taken, FIQ**

The counter counts each FIQ or virtual FIQ exception counted by [EXC_TAKEN](#) that is [Taken locally](#).

0x0088, **EXC_SMC, Exception taken, Secure Monitor Call**

The counter counts each Secure Monitor Call exception counted by [EXC_TAKEN](#).

The counter does not count SMC instructions that generate other exceptions, including Trap exceptions and Undefined Instruction exceptions.

0x008A, **EXC_HVC, Exception taken, Hypervisor Call**

The counter counts each Hypervisor Call exception counted by [EXC_TAKEN](#).

The counter does not count HVC instructions that are UNDEFINED.

0x008B, **EXC_TRAP_PABORT, Exception taken, Instruction Abort not Taken locally**

The counter counts each Instruction Abort exception counted by [EXC_TAKEN](#) that is not [Taken locally](#).

0x008C, **EXC_TRAP_DABORT, Exception taken, Data Abort or SError not Taken locally**

The counter counts each Data Abort, SError interrupt, or virtual SError interrupt exception counted by [EXC_TAKEN](#), that is not [Taken locally](#).

0x008D, **EXC_TRAP_OTHER**, Exception taken, other traps not Taken locally

The counter counts each exception counted by [EXC_TAKEN](#) that is not [Taken locally](#) and not counted as any of the following:

- Exception taken, Secure Monitor Call (EXC_SMC).
- Exception taken, Hypervisor Call (EXC_HVC).
- Exception taken, Instruction Abort not [Taken locally](#) (EXC_TRAP_PABORT).
- Exception taken, Data Abort or SError not [Taken locally](#) (EXC_TRAP_DABORT).
- Exception taken, IRQ not [Taken locally](#) (EXC_TRAP_IRQ).
- Exception taken, FIQ not [Taken locally](#) (EXC_TRAP_FIQ).

0x008E, **EXC_TRAP_IRQ**, Exception taken, IRQ not Taken locally

The counter counts each IRQ or virtual IRQ exception counted by [EXC_TAKEN](#) that is not [Taken locally](#).

0x008F, **EXC_TRAP_FIQ**, Exception taken, FIQ not Taken locally

The counter counts each FIQ or virtual FIQ exception counted by [EXC_TAKEN](#) that is not [Taken locally](#).

ASE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD

The counter counts each [architecturally executed](#) instruction counted by [ASE_SVE_INST_RETIRED](#) that is classified as an Advanced SIMD instruction.

The following instructions are counted as Advanced SIMD instructions:

- For AArch64 state:
 - The SIMD operations listed in Data processing - SIMD and floating-point.
 - If [FEAT_Crypto](#) is implemented, the Advanced SIMD Cryptographic instructions. See The Cryptographic Extension in AArch64 state.
 - Advanced SIMD loads and stores.
- For AArch32 state:
 - Instructions from [Advanced SIMD data-processing instructions](#).
 - If [FEAT_Crypto](#) is implemented, the `VMULL` (P64 variant) instruction. See [The Cryptographic Extension in AArch32 state](#).

This includes all instructions that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing instructions.
- Scalar floating-point data-processing instructions.

It is IMPLEMENTATION DEFINED whether Advanced SIMD scalar instructions are counted as Advanced SIMD instructions. This includes Advanced SIMD scalar instructions which operate on floating-point values.

ASE_RETIRED, Instruction architecturally executed, Advanced SIMD data processing

The counter counts each [architecturally executed](#) instruction counted by [ASE_SVE_RETIRED](#) that is classified as an Advanced SIMD data-processing instruction.

The following instructions are counted as Advanced SIMD data-processing instructions:

- For AArch64 state:
 - The SIMD operations listed in Data processing - SIMD and floating-point.
 - Advanced SIMD scalar instructions, including those which operate on floating-point values.
 - If [FEAT_Crypto](#) is implemented, the Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions. See The Cryptographic Extension in AArch64 state.
- For AArch32 state:

- Instructions from [Advanced SIMD data-processing instructions](#).
- If [FEAT_Crypto](#) is implemented, the `VMULL` (P64 variant) instruction. See [The Cryptographic Extension in AArch32 state](#).

This includes all instructions that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing instructions.
- Scalar floating-point data-processing instructions.
- Load or store instructions.
- Cryptographic data-processing instructions counted by `CRYPTO_RETIRED`. `CRYPTO_RETIRED` does not count the Cryptographic data-processing instructions included above.

ASE_SVE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD or SVE

The counter counts each [architecturally executed](#) instruction counted by `SE_INST_RETIRED` that is classified as an Advanced SIMD instruction or an SVE instruction.

See [ASE_INST_RETIRED](#) and `SVE_INST_RETIRED` for these classifications.

ASE_SVE_RETIRED, Instruction architecturally executed, Advanced SIMD data processing or SVE data processing

The counter counts each [architecturally executed](#) instruction counted by `INST_RETIRED` that is classified as an Advanced SIMD data-processing instruction or an SVE data-processing instruction.

See [ASE_RETIRED](#) and `SVE_RETIRED` for these classifications.

BL_IMMED_TAKEN_RETIRED, Branch instruction architecturally executed, direct branch with link, taken

The counter counts each branch counted by both [BL_TAKEN_RETIRED](#) and [BR_IMMED_TAKEN_RETIRED](#).

These are all direct branch with link instructions on the architecturally executed path, where the branch is taken.

BL_IND_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch with link, taken

The counter counts each branch counted by both [BL_TAKEN_RETIRED](#) and [BR_IND_TAKEN_RETIRED](#).

These are all indirect branch with link instructions on the architecturally executed path, where the branch is taken.

BL_TAKEN_RETIRED, Branch instruction architecturally executed, branch with link, taken

The counter counts each [Software change of the PC](#) counted by [PC_WRITE_RETIRED](#) that calls a subroutine and sets LR to return address.

These are all branch with link instructions on the architecturally executed path, where the branch is taken.

BRB_FILTRATE, Branch Record captured

The counter counts each valid Branch record captured in the branch record buffer.

Branch records that are not captured because they are removed by filtering are not counted.

When [BRB_FILTRATE](#) is generated for an exception or an exception return, it is an Exception-related event.

It is `CONSTRAINED UNPREDICTABLE` whether the counter counts Branch records injected by a BRB INJ instruction.

If counting this event causes unsigned overflow of the event counter counting the event and this in turn causes a BRBE freeze event then:

- The Branch record for the operation that generated the event will be generated and captured in the Branch record buffer.

- It is CONSTRAINED UNPREDICTABLE whether the Branch Record Buffer Extension generates Branch records for other operations in program order after the operation that generated the event that would otherwise be generated when generation of Branch records is not Paused.

Arm recommends that implementations minimize capture of additional branches.

BRNL_IMMED_TAKEN_RETIRED, Branch instruction architecturally executed, direct branch without link, taken

The counter counts each branch counted by both [BRNL_TAKEN_RETIRED](#) and [BR_IMMED_TAKEN_RETIRED](#).

These are all direct branch without link instructions on the architecturally executed path, where the branch is taken.

BRNL_INDNR_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch without link excluding procedure return, taken

The counter counts each branch counted by [BRNL_IND_TAKEN_RETIRED](#) that is not counted by [BR_RETURN_RETIRED](#).

These are all indirect branch without link excluding procedure return instructions on the architecturally executed path, where the branch is taken.

BRNL_IND_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch without link, taken

The counter counts each branch counted by both [BRNL_TAKEN_RETIRED](#) and [BR_IND_TAKEN_RETIRED](#).

These are all indirect branch without link instructions on the architecturally executed path, where the branch is taken.

BRNL_TAKEN_RETIRED, Branch instruction architecturally executed, branch without link, taken

The counter counts each [Software change of the PC](#) counted by [PC_WRITE_RETIRED](#) that is not counted by [BL_TAKEN_RETIRED](#).

These are all branch without link instructions on the architecturally executed path, where the branch is taken.

BR_COND_RETIRED, Branch instruction architecturally executed, conditional branch

The counter counts each [Software change of the PC](#) counted by [BR_RETIRED](#) that is a conditional branch.

These are all conditional branch instructions on the architecturally executed path.

In the A64 instruction set, conditional branches are:

- B.cond, CBNZ, CBZ, TBNZ, and TBZ.

This includes conditional branch instructions with the AL or NV condition code.

In the A32 instruction set, conditional branches are [Software change of the PC](#) instructions that have bits [31:29] of the instruction opcode not equal to 0b111.

In the T32 instruction set, conditional branches are B<c>, B<c>.W, [Software change of the PC](#) instructions executed in an IT block, CBNZ, and CBZ. Branch instruction encodings, including B{<c>} and B{<c>}.W, where <c> is either omitted or AL are only considered conditional branches when they are the last instruction in an IT block.

BR_COND_TAKEN_RETIRED, Branch instruction architecturally executed, conditional branch, taken

The counter counts each [Software change of the PC](#) counted by both [BR_COND_RETIRED](#) and [PC_WRITE_RETIRED](#).

These are all conditional branch instructions on the architecturally executed path, where the branch is taken.

BR_IMMED_SKIP_RETIRE, Branch instruction architecturally executed, immediate, not taken

The counter counts each instruction counted by both [BR_IMMED_RETIRE](#) and [BR_SKIP_RETIRE](#).

These are all direct branch instructions on the architecturally executed path, where the branch is not taken.

BR_IMMED_TAKEN_RETIRE, Branch instruction architecturally executed, immediate, taken

The counter counts each instruction counted by both [BR_IMMED_RETIRE](#) and [PC_WRITE_RETIRE](#).

These are all direct branch instructions on the architecturally executed path, where the branch is taken.

BR_INDNR_RETIRE, Branch instruction architecturally executed, indirect excluding procedure return

The counter counts each instruction counted by [BR_IND_RETIRE](#) that is not counted by [BR_RETURN_ANY_RETIRE](#).

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path.

BR_INDNR_SKIP_RETIRE, Branch instruction architecturally executed, indirect excluding procedure return, not taken

The counter counts each instruction counted by both [BR_INDNR_RETIRE](#) and [BR_SKIP_RETIRE](#).

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is not taken.

BR_INDNR_TAKEN_RETIRE, Branch instruction architecturally executed, indirect excluding procedure return, taken

The counter counts each instruction counted by both [BR_IND_RETIRE](#) and [PC_WRITE_RETIRE](#), that is not counted by [BR_RETURN_RETIRE](#).

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is taken.

BR_IND_RETIRE, Instruction architecturally executed, indirect branch

The counter counts each [architecturally executed](#) indirect branch instruction.

Indirect branch instructions are [Software change of the PC](#) instructions other than exception-generating instructions and immediate branch instructions. [Software change of the PC](#) has the same definition as for the [PC_WRITE_RETIRE](#) and [BR_SKIP_RETIRE](#) events. Immediate branch instructions are defined by the [BR_IMMED_RETIRE](#) event.

Note

Conditional branches are always counted, regardless of whether the branch is taken.

BR_IND_SKIP_RETIRE, Branch instruction architecturally executed, indirect, not taken

The counter counts each instruction counted by both [BR_IND_RETIRE](#) and [BR_SKIP_RETIRE](#).

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is not taken.

BR_IND_TAKEN_RETIRE, Branch instruction architecturally executed, indirect, taken

The counter counts each instruction counted by both [BR_IND_RETIRE](#) and [PC_WRITE_RETIRE](#).

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is taken.

BR_RETURN_ANY_RETIRED, Branch instruction architecturally executed, procedure return

The counter counts each instruction counted by [BR_IND_RETIRED](#) where if taken, the branch would be counted by [BR_RETURN_RETIRED](#).

These are all return instructions on the architecturally executed path.

BR_RETURN_SKIP_RETIRED, Branch instruction architecturally executed, procedure return, not taken

The counter counts each instruction counted by both [BR_RETURN_ANY_RETIRED](#) and [BR_SKIP_RETIRED](#).

These are all return instructions on the architecturally executed path, where the branch is not taken.

BR_SKIP_RETIRED, Branch instruction architecturally executed, not taken

The counter counts each conditional [Software change of the PC](#) instruction that is not taken.

This includes all:

- Conditional branch instructions.
- Conditional memory-reading instructions that explicitly write to the PC.
- Conditional data-processing instructions that explicitly write to the PC.
- Conditional exception return instructions.

These are the same instructions which, if unconditional, or conditional and taken, are counted by the [PC_WRITE_RETIRED](#) event.

It is IMPLEMENTATION DEFINED whether the following instructions are included as a [Software change of the PC](#), which the counter increments for:

- A conditional UNDEFINED instruction that does not generate an exception.
- A conditional exception-generating instruction, SVC, HVC, or SMC, that does not generate an exception.
- A conditional Context synchronization barrier instruction, ISB, that fails its condition code check.

Note

Many of these instructions can only be conditional in the AArch32 instruction sets.

The counter does not increment for exceptions other than those explicitly identified in these lists.

BR_UNCOND_RETIRED, Branch instruction architecturally executed, unconditional branch

The counter counts each [Software change of the PC](#) counted by [BR_RETIRED](#) that is not counted by [BR_COND_RETIRED](#).

These are all unconditional branch instructions on the architecturally executed path.

CRYPTO_RETIRED, Instruction architecturally executed, cryptographic data processing

The counter counts each [architecturally executed](#) instruction counted by [INST_RETIRED](#) that is classified as a cryptographic instruction.

The following instructions are counted as cryptographic data-processing instructions:

- In AArch64 state, instructions from The Cryptographic Extension in AArch64 state other than the include Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions.
- In AArch32 state, instructions from [The Cryptographic Extension in AArch32 state](#) other than `VMULL` (P64 variant).

DP_RETIRE, Instruction architecturally executed, integer data processing

The counter counts each [architecturally executed](#) instruction counted by [INST_RETIRE](#) that is classified as an integer data-processing instruction.

An integer data-processing instruction is any instruction that is not counted as any of the following:

- A load or store instruction, counted by [LD_RETIRE](#) or [ST_RETIRE](#).
- A Software change of PC instruction, counted by [PC_WRITE_RETIRE](#).
- A scalar floating-point data processing instruction, counted by [VFP_RETIRE](#).
- An Advanced SIMD or SVE data processing instruction, counted by [ASE_SVE_RETIRE](#).
- A cryptographic instruction, counted by [CRYPTO_RETIRE](#).

That is, the following instructions are counted as integer data-processing instructions:

- In AArch64 state:
 - Instructions from Data processing - immediate.
 - Instructions from Data processing - register.
 - Instructions from System register instructions.
 - Instructions from ‘Instructions with register argument’.
 - Instructions from System instructions other than ‘Memory-writing instructions’.
 - Instructions from Hint instructions.
 - Instructions from Floating-point move (register) that transfer data between a general-purpose register and a SIMD&FP register without conversion: [FMOV](#) (general).
 - Instructions from SIMD move that transfer data between a general-purpose register and an element or elements in a SIMD&FP register: [DUP](#) (general), [SMOV](#), [UMOV](#), and [INS](#) (general). This includes the aliases [MOV](#) (from general) and [MOV](#) (to general).
- In AArch32 state:
 - Instructions from [Data-processing instructions](#).
 - Instructions from [PSTATE and banked register access instructions](#).
 - Instructions from [Banked register access instructions](#).
 - Instructions from [Miscellaneous instructions](#) other than [ISB](#) and prefetches.
 - Instructions from [System register access instructions](#) other than [LDC](#) and [STC](#) instructions.
 - [VDUP](#) (general-purpose register).
 - All [VMOV](#) instructions that transfer data between a general-purpose register and a SIMD&FP register.
 - [VMRS](#) and [VMSR](#).

LDST_ANY_RETIRE, Instruction architecturally executed, load or store

The counter counts each [architecturally executed](#) instruction counted by [INST_RETIRE](#) that is classified as a Memory-reading instruction or a [Memory-writing instruction](#).

See [LD_ANY_RETIRE](#) and [ST_ANY_RETIRE](#) for these classifications.

If the [PRF_RETIRE](#) event is implemented and the prefetch instructions counted by [PRF_RETIRE](#) are not counted as Memory-reading instructions by [LD_RETIRE](#) and [LD_ANY_RETIRE](#), then it is IMPLEMENTATION DEFINED whether these instructions are counted by [LDST_ANY_RETIRE](#). Arm recommends that these instructions are counted by [LDST_ANY_RETIRE](#).

LD_ANY_RETIRE, Instruction architecturally executed, load

The counter counts each [architecturally executed](#) instruction counted by [LDST_ANY_RETIRE](#) that is classified as a Memory-reading instruction.

It is IMPLEMENTATION DEFINED whether the prefetch instructions counted by [PRF_RETIRE](#) are counted as Memory-reading instructions:

- If the prefetch instructions are counted as Memory-reading instructions, then they are counted by [LD_RETIRE](#) and [LD_ANY_RETIRE](#).
- Otherwise, if the prefetch instructions are not counted as Memory-reading instructions by [LD_ANY_RETIRE](#), then it is further IMPLEMENTATION DEFINED which one of the following applies:
 - They are counted as data-processing instructions.

- They are counted by [LDST_ANY_RETIRED](#).

Arm recommends that if a prefetch instruction is not implemented as a NOP and the [PRF_RETIRED](#) event is not implemented, then the instruction is counted as a Memory-reading instruction.

PRF_RETIRED, Instruction architecturally executed, prefetch

The counter counts each [architecturally executed](#) prefetch instruction counted by [INST_RETIRED](#).

SE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD, SVE, or SME

The counter counts each [architecturally executed](#) instruction counted by [INST_RETIRED](#) that is classified as an Advanced SIMD, SVE, or SME instruction.

See [ASE_SVE_INST_RETIRED](#) and [SME_INST_RETIRED](#) for these classifications.

SE_RETIRED, Instruction architecturally executed, Advanced SIMD, SVE, or SME data processing

The counter counts each [architecturally executed](#) instruction counted by [INST_RETIRED](#) that is classified as an Advanced SIMD, SVE, or SME data-processing instruction.

See [ASE_SVE_RETIRED](#) and [SME_RETIRED](#) for these classifications.

SIMD_INST_RETIRED, Instruction architecturally executed, SIMD

The counter counts each [architecturally executed](#) SIMD instruction counted by [INST_RETIRED](#) due to any of:

- An Advanced SIMD data-processing instruction. Advanced SIMD scalar data-processing instructions are not counted.
- An SVE SIMD data-processing instruction. SVE non-SIMD data-processing instructions are not counted.
- A structure load/store of one or more SIMD&FP registers.
- A load and replicate to one or more SIMD&FP registers.
- A scalar load/store of a SIMD&FP Q register or pair of Q registers.
- An SVE load/store.

It is IMPLEMENTATION DEFINED which [architecturally executed](#) SIMD instructions are counted in AArch32 state.

It is IMPLEMENTATION DEFINED whether scalar loads and stores to SIMD&FP registers other than those listed above are counted.

ST_ANY_RETIRED, Instruction architecturally executed, store

The counter counts each [architecturally executed](#) instruction counted by [LDST_ANY_RETIRED](#) that is classified as a [Memory-writing instruction](#).

TRB_TRIG, Trace buffer Trigger Event

The counter counts each trace buffer Trigger Event.

It is IMPLEMENTATION DEFINED whether this event can be counted by the PMU.

PMCEID0_EL0[46] reads as 1 if this event is implemented and can be counted by the PMU, and 0 otherwise.

TRB_WRAP, Trace buffer current write pointer wrapped

The counter counts each write made by the Trace Buffer Unit to the trace buffer that causes the trace buffer current write pointer to wrap to the trace buffer base pointer.

PMCEID0_EL0[44] reads as 1 if this event is implemented and 0 otherwise.

VFP_RETIRE, Instruction architecturally executed, scalar floating-point data processing

The counter counts each [architecturally executed](#) instruction counted by [INST_RETIRE](#) that is classified as a scalar floating-point instruction.

The following instructions are counted as scalar floating-point data-processing instructions:

- In AArch64 state:
 - The scalar floating-point instructions from Data processing - SIMD and floating-point. Only the scalar instructions are counted. SIMD floating-point instructions are not counted.
 - Instructions that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `FCVT{<mode>}`, `UCVTF`, and `SCVTF`.
- In AArch32 state:
 - Instructions from [Floating-point data-processing instructions](#).
 - Instructions that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `VCVT<mode>` (floating-point), `VCVT`, `VCVTT`, and `VCVTB`.

This includes all scalar instructions that operate on the SIMD&FP registers as floating-point values, except those that are counted as one of the following:

- SIMD scalar instructions.
- Integer data-processing instructions.
- Load or store instructions.

D7.3.2 Common microarchitectural events

The Common microarchitectural events are:

0x0001, L1I_CACHE_REFILL, Level 1 instruction cache refill

The counter counts each access counted by [L1I_CACHE](#) that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 instruction cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 instruction cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[1]` reads as 1 if this event is implemented and 0 otherwise.

0x0002, L1I_TLB_REFILL, Level 1 instruction TLB refill

The counter counts each [Instruction memory access](#) counted by [L1I_TLB](#) that causes a TLB refill of the Level 1 instruction TLB.

This includes each [Instruction memory access](#) that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable `TCR_ELx.EPDy` bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[2]` reads as 1 if this event is implemented and 0 otherwise.

0x0003, L1D_CACHE_REFILL, Level 1 data cache refill

The counter counts each access counted by [L1D_CACHE](#) that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 data cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[3]` reads as 1 if this event is implemented and 0 otherwise.

0x0004, **L1D_CACHE, Level 1 data cache access**

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that causes a cache access to at least the Level 1 data or unified cache.

Each access to a cache line is counted including the multiple accesses caused by single instructions such as `LDM` or `STM`. Each access to other Level 1 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

It is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted. For example, whether the cached data is held in an exclusive or shared state.

When the [L1D_CACHE_RW](#) event is implemented:

- If the [L1D_CACHE_PRFM](#) event is implemented, accesses to the Level 1 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the [L1D_CACHE_HWPRF](#) event is implemented, accesses to the Level 1 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the [L1D_CACHE_RW](#) event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 1 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[4]` reads as 1 if this event is implemented and 0 otherwise.

0x0005, **L1D_TLB_REFILL, Level 1 data TLB refill**

The counter counts each access counted by [L1D_TLB](#) that causes a TLB refill of the Level 1 data or unified TLB.

This includes each access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[5]` reads as 1 if this event is implemented and 0 otherwise.

0x0010, **BR_MIS_PRED**, Branch instruction speculatively executed, mispredicted or not predicted

The counter counts each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the branch prediction resources and that relates to instructions that the branch prediction resources are capable of predicting.

If no program-flow prediction resources are implemented, Arm recommends that the counter counts all branches that are not taken.

`PMCEID0_EL0[16]` reads as 1 if this event is implemented and 0 otherwise.

0x0011, **CPU_CYCLES**, Cycle

The counter increments on every cycle when the PE is not in `WFI` or `WFE` state.

When the PE is in `WFI` or `WFE` state, it is IMPLEMENTATION SPECIFIC whether the counter counts.

Note

Unlike `PMCCNTR` or `PMCCNTR_EL0`, this count is not affected by the cycle counter controls:

- The counter is not incremented in prohibited regions, so is not affected by `PMCR.DP` or `PMCR_EL0.DP`.
 - The counter increments on every cycle, regardless of the value of `PMCR.D` or `PMCR_EL0.D`.
 - The counter is reset when event counters are reset by `PMCR.P` and `PMCR_EL0.P`, never by `PMCR.C` or `PMCR_EL0.C`.
-

In a multithreaded implementation:

- If the Effective value of `PMEVTYPER<n>_EL0.MT` for the event counter is 0, then the counter counts each [Processor cycle](#) when the counting PE is active.
- If the Effective value of `PMEVTYPER<n>_EL0.MT` for the event counter is 1, then the counter counts each [Processor cycle](#).

`PMCEID0_EL0[17]` reads as 1 if this event is implemented and 0 otherwise.

0x0012, **BR_PRED**, Predictable branch instruction speculatively executed

The counter counts each branch or other change in the program flow that the branch prediction resources are capable of predicting.

If all branches are subject to prediction, for example a BTB or BTAC, then all branches are predictable branches.

If branches are decoded before the predictor, so that the branch prediction logic dynamically predicts only some branches, for example conditional and indirect branches, then it is IMPLEMENTATION DEFINED whether other branches are counted as predictable branches. Arm recommends that all branches are counted.

An implementation might include other structures that predict branches, such as a loop buffer that predicts short backwards direct branches as taken. Each execution of such a branch is a predictable branch. Terminating the loop might generate a misprediction event that is counted by [BR_MIS_PRED](#).

If no program-flow prediction resources are implemented, this event is optional, but Arm recommends that [BR_PRED](#) counts all branches.

It is IMPLEMENTATION DEFINED when the branch is counted. Arm recommends that it is counted when the branch is resolved, that is, at the same point in the instruction pipeline as when the [BR_MIS_PRED](#) event would be counted if the branch resolves as mispredicted.

PMCEID0_EL0[18] reads as 1 if this event is implemented and 0 otherwise.

0x0013, **MEM_ACCESS**, Data memory access

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that the PE makes.

The counter increments whether the access results in an access to a Level 1 data or unified cache, a Level 2 data or unified cache, or neither of these.

The number of accesses generated by each instruction is IMPLEMENTATION DEFINED.

The counter does not count:

- [Instruction memory accesses](#).
- Translation table walks.
- Write-back from any cache.
- Refilling of any cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

PMCEID0_EL0[19] reads as 1 if this event is implemented and 0 otherwise.

0x0014, **L1I_CACHE**, Level 1 instruction cache access

The counter counts each [Instruction memory access](#) to at least the Level 1 instruction or unified cache.

Each [Instruction memory access](#) to other Level 1 instruction or unified memory structures, such as refill buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

It is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted. For example, whether the cached data is held in an exclusive or shared state.

When the [L1I_CACHE_RD](#) event is implemented:

- If the [L1I_CACHE_PRFM](#) event is implemented, accesses to the Level 1 instruction cache due to a prefetch instruction are counted. Otherwise, these are not counted.
- If the [L1I_CACHE_HWPRF](#) event is implemented, accesses to the Level 1 instruction cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the [L1I_CACHE_RD](#) event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 1 instruction cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

PMCEID0_EL0[20] reads as 1 if this event is implemented and 0 otherwise.

0x0015, **L1D_CACHE_WB, Level 1 data cache write-back**

The counter counts each write-back of data from the Level 1 data or unified cache to outside of the Level 1 caches.

For example:

- A write-back of a cache line to a Level 2 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 1 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 1 data cache to outside of Level 1 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 caches or memory.
- Writes that write through the Level 1 data cache to outside of the Level 1 caches.

A write-back is attributable to one of the following, and it is IMPLEMENTATION DEFINED which:

- The PE or other agent that generated the request that caused the write-back. This might not be the same agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that last accessed the data when it was in the cache. If the data has not been accessed since being allocated, then the PE or other agent that caused the data to be allocated into the cache.

If the cache is shared, then the write-back event is counted only if it is attributable to the PE counting the event.

If the cache is not shared, then the event is counted by the PE that the cache is attached to, even if the write-back is not attributable to that PE. For example, a requestor outside of the PE made a coherency request that resulted in write-back.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

PMCEID0_EL0[21] reads as 1 if this event is implemented and 0 otherwise.

0x0016, **L2D_CACHE, Level 2 data cache access**

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that causes a cache access to at least the Level 2 data or unified cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Level 2 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

It is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted. For example, whether the cached data is held in an exclusive or shared state.

When the [L2D_CACHE_RW](#) event is implemented:

- If the [L2D_CACHE_PRFM](#) event is implemented, accesses to the Level 2 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the [L2D_CACHE_HWPRF](#) event is implemented, accesses to the Level 2 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the [L2D_CACHE_RW](#) event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 2 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[22]` reads as 1 if this event is implemented and 0 otherwise.

0x0017, **L2D_CACHE_REFILL, Level 2 data cache refill**

The counter counts each access counted by **L2D_CACHE** that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 and Level 2 caches, even if the data is ultimately not allocated into the Level 2 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 data cache that cause a refill that is satisfied by fetching data from memory, a Level 3 cache, or a Level 2 cache of another PE.
- Refills of and write-backs from any Level 1 data, instruction, or unified cache of this PE that cause a refill from outside of the Level 1 and Level 2 caches of this PE.
- Accesses to the Level 2 data cache that cause a refill of a Level 1 cache of this PE from outside of the Level 1 and Level 2 caches of this PE, even if there is no refill of the Level 2 data cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[23]` reads as 1 if this event is implemented and 0 otherwise.

0x0018, **L2D_CACHE_WB, Level 2 data cache write-back**

The counter counts each write-back of data from the Level 2 data or unified cache to outside of the Level 1 and Level 2 caches.

For example:

- A write-back of a cache line to a Level 3 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 2 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 2 data cache to outside of Level 1 and Level 2 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 and Level 2 caches or memory.
- Writes that write through the Level 2 data cache to outside of the Level 1 and Level 2 caches.
- Transfers of data from the Level 2 data cache to a Level 1 cache to satisfy a refill of the other cache.

A write-back is attributable to one of the following, and it is IMPLEMENTATION DEFINED which:

- The PE or other agent that generated the request that caused the write-back. This might not be the same agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that last accessed the data when it was in the cache. If the data has not been accessed since being allocated, then the PE or other agent that caused the data to be allocated into the cache.

If the cache is shared, then the write-back event is counted only if it is attributable to the PE counting the event.

If the cache is not shared, then the event is counted by the PE that the cache is attached to, even if the write-back is not attributable to that PE. For example, a requestor outside of the PE made a coherency request that resulted in write-back.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

PMCEID0_EL0[24] reads as 1 if this event is implemented and 0 otherwise.

0x0019, **BUS_ACCESS**, Bus access

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that accesses outside of the boundary of the PE and its closely-coupled caches.

Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. Where an implementation has multiple buses at this boundary, this event counts the sum of accesses across all buses.

A bus access is part of a bus transaction. The exact nature of a bus transaction is IMPLEMENTATION DEFINED, but for the purposes of event monitoring consists of a single access comprising one or more cycles, or beats, when the transaction occupies the bus. The [BUS_ACCESS](#) event counts each beat of each transaction. That is, each bus cycle counted by [BUS_CYCLES](#) for which the bus is active.

Bus transactions include refills of and write-backs from data, instruction, and unified caches. Whether bus transactions include operations that use the bus but do not explicitly transfer data is IMPLEMENTATION DEFINED.

An Unattributable bus transaction occurs when a requester outside the PE makes a request that results in a bus access, for example, a coherency request.

If a bus supports multiple accesses per cycle, for example through multiple channels, the counter increments once for each channel that is active on a cycle, and so it might increment by more than one in any given cycle.

The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

PMCEID0_EL0[25] reads as 1 if this event is implemented and 0 otherwise.

0x001A, **MEMORY_ERROR**, Local memory error

The counter counts each occurrence of a memory error signaled by a memory closely coupled to this PE.

The definition of local memories is IMPLEMENTATION DEFINED but includes caches, tightly-coupled memories, and TLB arrays.

Memory error refers to a physical error detected by the hardware, such as a parity or ECC error. It includes errors that are correctable and those that are not. It does not include errors as defined in the architecture, such as MMU faults.

PMCEID0_EL0[26] reads as 1 if this event is implemented and 0 otherwise.

0x001B, **INST_SPEC**, Operation speculatively executed

The counter counts each [speculatively executed](#) operation.

This includes operations that are subsequently not architecturally executed.

PMCEID0_EL0[27] reads as 1 if this event is implemented and 0 otherwise.

0x001D, **BUS_CYCLES, Bus cycle**

The counter increments on every cycle of the interface at the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED.

Note

If the implementation clocks the external memory interface at the same rate as the processor hardware, then the counter counts every cycle.

PMCEID0_EL0[29] reads as 1 if this event is implemented and 0 otherwise.

0x001F, **L1D_CACHE_ALLOCATE, Level 1 data cache allocation without refill**

The counter counts each [Memory-write operation](#) that writes an entire line into the Level 1 data or unified cache without fetching data from outside the Level 1 data or unified cache.

These are allocations of cache lines in the Level 1 data or unified cache that are not refills counted by [L1D_CACHE_REFILL](#). For example:

- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only [Memory-write operations](#) Attributable to the PE counting the event, and, if the Effective value of `PMEVTPER<n>_EL0.MT` for the counter is 1, other PEs in the multithreaded implementation.

PMCEID0_EL0[31] reads as 1 if this event is implemented and 0 otherwise.

0x0020, **L2D_CACHE_ALLOCATE, Level 2 data cache allocation without refill**

The counter counts each [Memory-write operation](#) that writes an entire line into the Level 2 data or unified cache without fetching data from outside the Level 2 data or unified cache.

These are allocations of cache lines in the Level 2 data or unified cache that are not refills counted by [L2D_CACHE_REFILL](#). For example:

- A write-back of an entire cache line from a Level 1 cache to the Level 2 data cache.
- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only [Memory-write operations](#) Attributable to the PE counting the event, and, if the Effective value of `PMEVTPER<n>_EL0.MT` for the counter is 1, other PEs in the multithreaded implementation.

PMCEID1_EL0[0] reads as 1 if this event is implemented and 0 otherwise.

0x0022, **BR_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted**

The counter counts each branch instruction counted by [BR_RETIRED](#) that was not correctly predicted.

If no program-flow prediction resources are implemented, this event counts all retired not-taken branches.

PMCEID1_EL0[2] reads as 1 if this event is implemented and 0 otherwise.

0x0023, **STALL_FRONTEND, No operation sent for execution due to the frontend**

The counter counts each cycle counted by [CPU_CYCLES](#) where no Attributable instruction or operation was sent for execution and there was no Attributable instruction or operation available to dispatch for the PE from the frontend.

The division between frontend and backend is IMPLEMENTATION DEFINED. All [STALL](#) events are counted at the same point in the pipeline.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts only if the stall condition is true for all the PEs in the multithreaded processor.

Note

- For a simplified pipeline model of Fetch-Decode-Issue-Execute-Retire, Arm recommends that the events are counted when instructions are dispatched from Decode to Issue.
 - In a single cycle, both the [STALL_BACKEND](#) and [STALL_FRONTEND](#) events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.
-

`PMCEID1_EL0[3]` reads as 1 if this event is implemented and 0 otherwise.

0x0024, **STALL_BACKEND**, No operation sent for execution due to the backend

The counter counts each cycle counted by [CPU_CYCLES](#) where Attributable instructions or operations for the PE are available to dispatch from the frontend to the backend, but no Attributable instruction or operation is sent for execution because the backend is unable to accept any of the instructions or operations available for the PE.

For example, the backend might be unable to accept operations because of a resource conflict or non-availability.

It is IMPLEMENTATION DEFINED whether the counter also counts each cycle counted by [CPU_CYCLES](#) where no Attributable instructions or operations for the PE are available to dispatch from the frontend and the backend is unable to accept any instructions or operations for the PE.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see [STALL_FRONTEND](#). All [STALL](#) events are counted at the same point in the pipeline.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts only if the stall condition is true for all the PEs in the multithreaded processor.

Note

In a single cycle, both the [STALL_BACKEND](#) and [STALL_FRONTEND](#) events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.

`PMCEID1_EL0[4]` reads as 1 if this event is implemented and 0 otherwise.

0x0025, **L1D_TLB**, Level 1 data TLB access

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that causes a TLB access to at least the Level 1 data or unified TLB.

Each access to a TLB entry is counted including multiple accesses caused by single instructions such as LDM or STM.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[5]` reads as 1 if this event is implemented and 0 otherwise.

0x0026, **L1I_TLB**, Level 1 instruction TLB access

The counter counts each [Instruction memory access](#) that causes a TLB access to at least the Level 1 instruction TLB.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT`

for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[6]` reads as 1 if this event is implemented and 0 otherwise.

0x0027, **L2I_CACHE**, Level 2 instruction cache access

The counter counts each [Instruction memory access](#) to at least the Level 2 instruction or unified cache.

Each [Instruction memory access](#) to other Level 2 instruction or unified memory structures, such as refill buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

It is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted. For example, whether the cached data is held in an exclusive or shared state.

When the [L2I_CACHE_RD](#) event is implemented:

- If the [L2I_CACHE_PRFM](#) event is implemented, accesses to the Level 2 instruction cache due to a prefetch instruction are counted. Otherwise, these are not counted.
- If the [L2I_CACHE_HWPRF](#) event is implemented, accesses to the Level 2 instruction cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the [L2I_CACHE_RD](#) event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 2 instruction cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[7]` reads as 1 if this event is implemented and 0 otherwise.

0x0028, **L2I_CACHE_REFILL**, Level 2 instruction cache refill

The counter counts each access counted by [L2I_CACHE](#) that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 and Level 2 caches, even if the data is ultimately not allocated into the Level 2 instruction cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 instruction cache that cause a refill that is satisfied by fetching data from memory, a Level 3 cache, or a Level 2 cache of another PE.
- Refills of any Level 1 instruction or unified cache of this PE that cause a refill from outside of the Level 1 and Level 2 caches of this PE.
- Accesses to the Level 2 instruction cache that cause a refill of a Level 1 cache of this PE from outside of the Level 1 and Level 2 caches of this PE, even if there is no refill of the Level 2 instruction cache.

The counter does not count accesses that miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other

cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[8]` reads as 1 if this event is implemented and 0 otherwise.

0x0029, **L3D_CACHE_ALLOCATE, Level 3 data cache allocation without refill**

The counter counts each [Memory-write operation](#) that writes an entire line into the Level 3 data or unified cache without fetching data from outside the Level 3 data or unified cache.

These are allocations of cache lines in the Level 3 data or unified cache that are not refills counted by [L3D_CACHE_REFILL](#). For example:

- A write-back of an entire cache line from a Level 2 cache to the Level 3 data cache.
- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only [Memory-write operations](#) Attributable to the PE counting the event, and, if the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, other PEs in the multithreaded implementation.

`PMCEID1_EL0[9]` reads as 1 if this event is implemented and 0 otherwise.

0x002A, **L3D_CACHE_REFILL, Level 3 data cache refill**

The counter counts each access counted by [L3D_CACHE](#) that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 to Level 3 caches, even if the data is ultimately not allocated into the Level 3 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 3 data cache that cause a refill that is satisfied by fetching data from memory, a lower level cache, or a Level 3 cache of another PE.
- Refills of and write-backs from any Level 1 or Level 2 data, instruction, or unified caches of this PE that cause a refill from outside of the Level 1 to Level 3 caches of this PE.
- Accesses to the Level 3 data cache that cause a refill of a Level 2 cache of this PE from outside of the Level 1 to Level 3 caches of this PE, even if there is no refill of the Level 3 data cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[10]` reads as 1 if this event is implemented and 0 otherwise.

0x002B, **L3D_CACHE, Level 3 data cache access**

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that causes a cache access to at least the Level 3 data or unified cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Level 3 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

It is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted. For example, whether the cached data is held in an exclusive or shared state.

When the **L3D_CACHE_RW** event is implemented:

- If the **L3D_CACHE_PRFM** event is implemented, accesses to the Level 3 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the **L3D_CACHE_HWPRF** event is implemented, accesses to the Level 3 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the **L3D_CACHE_RW** event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 3 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of **PMEVTYPER<n>_EL0.MT** for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of **PMEVTYPER<n>_EL0.MT** for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of **PMEVTYPER<n>_EL0.MT**.

PMCEID1_EL0[11] reads as 1 if this event is implemented and 0 otherwise.

0x002C, **L3D_CACHE_WB, Level 3 data cache write-back**

The counter counts each write-back of data from the Level 3 data or unified cache to outside of the Level 1 to Level 3 caches.

For example:

- A write-back of a cache line to a lower level cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 3 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 3 data cache to outside of Level 1 to Level 3 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 to Level 3 caches or memory.
- Writes that write through the Level 3 data cache to outside of the Level 1 to Level 3 caches.
- Transfers of data from the Level 3 data cache to a Level 1 or Level 2 cache to satisfy a refill of the other cache.

A write-back is attributable to one of the following, and it is IMPLEMENTATION DEFINED which:

- The PE or other agent that generated the request that caused the write-back. This might not be the same agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that last accessed the data when it was in the cache. If the data has not been accessed since being allocated, then the PE or other agent that caused the data to be allocated into the cache.

If the cache is shared, then the write-back event is counted only if it is attributable to the PE counting the event.

If the cache is not shared, then the event is counted by the PE that the cache is attached to, even if the write-back is not attributable to that PE. For example, a requestor outside of the PE made a coherency request that resulted in write-back.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

PMCEID1_EL0[12] reads as 1 if this event is implemented and 0 otherwise.

0x002D, L2D_TLB_REFILL, Level 2 data TLB refill

The counter counts each access counted by L2D_TLB that causes a TLB refill of the Level 2 data or unified TLB, or any Level 1 data, instruction, or unified TLBs of this PE.

This includes each access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[13] reads as 1 if this event is implemented and 0 otherwise.

0x002E, L2I_TLB_REFILL, Level 2 instruction TLB refill

The counter counts each Instruction memory access counted by L2I_TLB that causes a TLB refill of the Level 2 instruction TLB, or any Level 1 data, instruction, or unified TLBs of this PE.

This includes each Instruction memory access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[14] reads as 1 if this event is implemented and 0 otherwise.

0x002F, **L2D_TLB, Level 2 data TLB access**

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that causes a TLB access to at least the Level 2 data or unified TLB.

Each access to a TLB entry is counted including refills of Level 1 TLBs.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[15]` reads as 1 if this event is implemented and 0 otherwise.

0x0030, **L2I_TLB, Level 2 instruction TLB access**

The counter counts each [Instruction memory access](#) that causes a TLB access to at least the Level 2 instruction TLB.

Each access to a TLB entry is counted including refills of Level 1 TLBs.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[16]` reads as 1 if this event is implemented and 0 otherwise.

0x0031, **REMOTE_ACCESS, Access to a remote device**

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that causes an access to a remote device.

That is, a socket that does not contain the PE.

System topology is IMPLEMENTATION DEFINED. This means that it is IMPLEMENTATION DEFINED which systems are defined as multi-socket systems, and, in systems defined as multi-socket systems, which components are defined as being in the same or different sockets. Arm recommends that devices where an access through that device incurs a significant latency penalty compared to other accesses are treated as being in a different socket.

For example, in a system comprising multiple integrated circuits in a multi-chip module, an access to a different integrated circuit in the same module might be treated as an access to another socket, even though the multi-chip module is physically connected to a single socket at the motherboard. However, in another system with many such multi-chip modules, an access to a different integrated system in the same module might be treated as an access to the same socket because an access to an integrated circuit on a different module has much higher latency.

It is IMPLEMENTATION DEFINED whether an access that causes a snoop into a different socket but does not return data from or pass data to the remote socket is counted.

`PMCEID1_EL0[17]` reads as 1 if this event is implemented and 0 otherwise.

0x0032, **LL_CACHE, Last level cache access**

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) that causes a cache access to at least the Last level cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Last level data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

It is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted. For example, whether the cached data is held in an exclusive or shared state.

When the [LL_CACHE_RW](#) event is implemented:

- If the [LL_CACHE_PRFM](#) event is implemented, accesses to the Last level cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the [LL_CACHE_HWPRF](#) event is implemented, accesses to the Last level cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the [LL_CACHE_RW](#) event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Last level cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[18]` reads as 1 if this event is implemented and 0 otherwise.

0x0033, [LL_CACHE_MISS](#), Last level cache miss

If the [LL_CACHE_RW](#) event is implemented, the counter counts each access counted by [LL_CACHE_RW](#) that is not completed by the Last level cache.

If the [LL_CACHE_RW](#) event is not implemented, the counter counts each access counted by [LL_CACHE](#) that is not completed by the Last level cache.

That is, the access is one of the following:

- A ‘Memory-read operation’ that does not return data from the Last level cache.
- A [Memory-write operation](#) that does not update the Last level cache.

The counter does not count operations that are completed by a cache above the Last level cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[19]` reads as 1 if this event is implemented and 0 otherwise.

0x0034, [DTLB_WALK](#), Data TLB access with at least one translation table walk

The counter counts each access counted by [L1D_TLB](#) that causes a refill of a data or unified TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

It is IMPLEMENTATION DEFINED whether accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable `TCR_ELx.EPDy` bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[20]` reads as 1 if this event is implemented and 0 otherwise.

0x0035, **ITLB_WALK**, Instruction TLB access with at least one translation table walk

The counter counts each access counted by `L1I_TLB` that causes a refill of an instruction TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

It is IMPLEMENTATION DEFINED whether accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable `TCR_ELx.EPDy` bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[21]` reads as 1 if this event is implemented and 0 otherwise.

0x0036, **LL_CACHE_RD**, Last level cache access, read

If the `LL_CACHE_RW` event is implemented, the counter counts each access counted by `LL_CACHE_RW` that is a ‘Memory-read operation’.

If the `LL_CACHE_RW` event is not implemented, the counter counts each access counted by `LL_CACHE` that is a ‘Memory-read operation’.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[22]` reads as 1 if this event is implemented and 0 otherwise.

0x0037, **LL_CACHE_MISS_RD**, Last level cache miss, read

The counter counts each access counted by both `LL_CACHE_MISS` and `LL_CACHE_RD`.

That is, every miss of the Last level cache counted by `LL_CACHE_MISS` that is caused by a ‘Memory-read operation’.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is

shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[23]` reads as 1 if this event is implemented and 0 otherwise.

0x0038, **REMOTE_ACCESS_RD, Access to a remote device, read**

The counter counts each access counted by `REMOTE_ACCESS` that is a 'Memory-read operation'.

`PMCEID1_EL0[24]` reads as 1 if this event is implemented and 0 otherwise.

0x0039, **L1D_CACHE_LMISS_RD, Level 1 data cache long-latency read miss**

The counter counts each 'Memory-read operation' to the Level 1 data or unified cache counted by `L1D_CACHE_RD` that incurs additional latency because it returns data from outside of the Level 1 data or unified cache of this PE.

The event indicates to software that the access missed in the Level 1 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 1 caches that the access looks up in and results in an access being made to a Level 2 cache or elsewhere beyond the Level 1 data or unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID1_EL0[25]` reads as 1 if this event is implemented and 0 otherwise.

0x003A, **OP_RETIRED, Micro-operation architecturally executed**

The counter counts each operation counted by `OP_SPEC` that would be executed in a Simple sequential execution of the program.

`PMCEID1_EL0[26]` reads as 1 if this event is implemented and 0 otherwise.

0x003B, **OP_SPEC, Micro-operation speculatively executed**

The counter counts each operation `speculatively executed` executed by the PE.

This includes operations that would not be executed in a Simple sequential execution of the program.

`PMCEID1_EL0[27]` reads as 1 if this event is implemented and 0 otherwise.

0x003C, **STALL, No operation sent for execution**

The counter counts each cycle counted by [CPU_CYCLES](#) where no operation was sent for execution.

On a multithreaded implementation:

- If the Effective value of `PMEVTPER<n>_EL0.MT` is 0, then the counter counts cycles when the PE is active and no operation for the PE is sent for execution, even if operations Attributable to other PEs in the multithreaded implementation are sent for execution in that cycle. The counter does not count cycles when the PE is not active.
- If the Effective value of `PMEVTPER<n>_EL0.MT` is 1, then the counter counts all cycles when no instructions or operations for any PE in the multithreaded implementation were sent for execution.

All [STALL](#) events are counted at the same point in the pipeline.

`PMCEID1_EL0[28]` reads as 1 if this event is implemented and 0 otherwise.

0x003D, **STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend**

The counter counts each Slot counted by [STALL_SLOT](#) where an Attributable instruction or operation for the PE is available on the Slot to dispatch from the frontend to the backend, but is not sent for execution because the backend is unable to accept the available instruction or operation for the PE.

It is IMPLEMENTATION DEFINED whether the counter also counts each Slot counted by [STALL_SLOT](#) where no Attributable instruction or operation for the PE is available on the Slot to dispatch from the frontend and the backend is unable to accept any instruction or operation for the PE.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see [STALL_FRONTEND](#). All [STALL](#) events are counted at the same point in the pipeline. The maximum value by which [STALL_SLOT_BACKEND](#) can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`. For more information, see [STALL_SLOT](#).

Note

In a single cycle, the sum of values counted by [STALL_SLOT_BACKEND](#) and [STALL_SLOT_FRONTEND](#) might be greater-than `slots`, if both the backend is unable to accept some operations and there are fewer-than `slots` operations available to dispatch from the frontend.

`PMCEID1_EL0[29]` reads as 1 if this event is implemented and 0 otherwise.

0x003E, **STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend**

The counter counts each Slot counted by [STALL_SLOT](#) where no Attributable instruction or operation was sent for execution and there was no Attributable instruction or operation available to dispatch for the PE from the frontend for the Slot.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see [STALL_FRONTEND](#). All [STALL](#) events are counted at the same point in the pipeline. The maximum value by which [STALL_SLOT_FRONTEND](#) can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`. For more information, see [STALL_SLOT](#).

Note

In a single cycle, the sum of values counted by [STALL_SLOT_BACKEND](#) and [STALL_SLOT_FRONTEND](#) might be greater-than `slots`, if both the backend is unable to accept some operations and there are fewer-than `slots` operations available to dispatch from the frontend.

`PMCEID1_EL0[30]` reads as 1 if this event is implemented and 0 otherwise.

0x003F, **STALL_SLOT, No operation sent for execution on a Slot**

The counter increments by the number of instruction or operation Slots that were not occupied by an instruction or operation Attributable to the PE on each cycle.

All **STALL** events are counted at the same point in the pipeline. The maximum value by which **STALL_SLOT** can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`. The definition of a Slot is IMPLEMENTATION DEFINED. The formula $\text{STALL_SLOT} \div (\text{CPU_CYCLES} \times \text{slots})$ gives the utilization of the Slots of the processor by Attributable instruction or operations of this PE. Each Slot holds at most one instruction or operation each cycle.

On a multithreaded implementation:

- If the Effective value of **PMEVTYPER**<n>_EL0.MT is 0, then the counter counts Slots occupied by an instruction or operation Attributable to other PEs of the multithreaded implementation only when the PE was active in that cycle. The counter does not count Slots on cycles when the PE was not active.
- If the Effective value of **PMEVTYPER**<n>_EL0.MT is 1, then, for every cycle, the counter counts all Slots not occupied by any instruction or operation for any PE of the multithreaded implementation.

PMCEID1_EL0[31] reads as 1 if this event is implemented and 0 otherwise.

0x0040, **L1D_CACHE_RD, Level 1 data cache access, read**

If the **L1D_CACHE_RW** event is implemented, the counter counts each access counted by **L1D_CACHE_RW** that is a ‘Memory-read operation’.

If the **L1D_CACHE_RW** event is not implemented, the counter counts each access counted by **L1D_CACHE** that is a ‘Memory-read operation’.

If the cache is shared and the Effective value of **PMEVTYPER**<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of **PMEVTYPER**<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of **PMEVTYPER**<n>_EL0.MT.

0x0041, **L1D_CACHE_WR, Level 1 data cache access, write**

If the **L1D_CACHE_RW** event is implemented, the counter counts each access counted by **L1D_CACHE_RW** that is a Memory-write operation.

If the **L1D_CACHE_RW** event is not implemented, the counter counts each access counted by **L1D_CACHE** that is a Memory-write operation.

If the cache is shared and the Effective value of **PMEVTYPER**<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of **PMEVTYPER**<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of **PMEVTYPER**<n>_EL0.MT.

0x0042, **L1D_CACHE_REFILL_RD, Level 1 data cache refill, read**

The counter counts each access counted by both **L1D_CACHE_RD** and **L1D_CACHE_REFILL**.

That is, every refill of the Level 1 data or unified cache counted by **L1D_CACHE_REFILL** that is caused by a ‘Memory-read operation’.

If the cache is shared and the Effective value of **PMEVTYPER**<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of **PMEVTYPER**<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of **PMEVTYPER**<n>_EL0.MT.

0x0043, **L1D_CACHE_REFILL_WR, Level 1 data cache refill, write**

The counter counts each access counted by both **L1D_CACHE_REFILL** and **L1D_CACHE_WR**.

That is, every refill of the Level 1 data or unified cache counted by [L1D_CACHE_REFILL](#) that is caused by a [Memory-write operation](#).

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x0044, **L1D_CACHE_REFILL_INNER, Level 1 data cache refill, inner**

The counter counts each access counted by [L1D_CACHE_REFILL](#) that generates a refill satisfied by transfer from another cache inside of the immediate cluster.

Note

The boundary between inner and outer is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x0045, **L1D_CACHE_REFILL_OUTER, Level 1 data cache refill, outer**

The counter counts each access counted by [L1D_CACHE_REFILL](#) that generates a refill satisfied by transfer from outside of the immediate cluster.

Note

The boundary between inner and outer is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x0046, **L1D_CACHE_WB_VICTIM, Level 1 data cache write-back, victim**

The counter counts each write-back counted by [L1D_CACHE_WB](#) that occurs because of a capacity eviction due to a line being allocated into the cache..

It is IMPLEMENTATION DEFINED whether this includes capacity evictions due to an agent other the PE counting the event. For example, a stashing request from outside of the PE, or an allocation due to another PE that shares the cache.

In all cases, the event is counted only if the eviction is also an [L1D_CACHE_WB](#) event. This means that, if the cache is shared, the eviction is counted only if it is attributable to the PE counting the event, as defined by [L1D_CACHE_WB](#).

0x0047, **L1D_CACHE_WB_CLEAN, Level 1 data cache write-back, cleaning and coherency**

The counter counts each write-back counted by [L1D_CACHE_WB](#) that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction..

It is IMPLEMENTATION DEFINED whether the transfer of a dirty cache line from the Level 1 data cache of this PE to the data cache of another PE due to a hardware coherency operation is counted when the dirty cache line is not also written back to a Level 2 cache or memory.

In all cases, the event is counted only if the write-back is also an [L1D_CACHE_WB](#) event. This means that, if the cache is shared, the write-back is counted only if it is attributable to the PE counting the event, as defined by [L1D_CACHE_WB](#).

0x0048, **L1D_CACHE_INVALID, Level 1 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 1 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 1 data or unified cache invalidates a line in the Level 1 data or unified cache.

The counter does not count locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 1 data cache is shared with other PEs. This event does not follow the general rule of Level 1 data cache events of only counting Attributable events.

0x004C, **L1D_TLB_REFILL_RD, Level 1 data TLB refill, read**

The counter counts each access counted by both [L1D_TLB_RD](#) and [L1D_TLB_REFILL](#).

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

0x004D, **L1D_TLB_REFILL_WR, Level 1 data TLB refill, write**

The counter counts each access counted by both [L1D_TLB_REFILL](#) and [L1D_TLB_WR](#).

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

0x004E, L1D_TLB_RD, Level 1 data TLB access, read

If the [L1D_TLB_RW](#) event is implemented, the counter counts each access counted by [L1D_TLB_RW](#) that is due to a demand 'Memory-read operation'.

If the [L1D_TLB_RW](#) event is not implemented, the counter counts each access counted by [L1D_TLB](#) that is due to a 'Memory-read operation'.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x004F, L1D_TLB_WR, Level 1 data TLB access, write

If the [L1D_TLB_RW](#) event is implemented, the counter counts each access counted by [L1D_TLB_RW](#) that is due to a demand [Memory-write operation](#).

If the [L1D_TLB_RW](#) event is not implemented, the counter counts each access counted by [L1D_TLB](#) that is due to a [Memory-write operation](#).

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x0050, L2D_CACHE_RD, Level 2 data cache access, read

If the [L2D_CACHE_RW](#) event is implemented, the counter counts each access counted by [L2D_CACHE_RW](#) that is a 'Memory-read operation'.

If the [L2D_CACHE_RW](#) event is not implemented, the counter counts each access counted by [L2D_CACHE](#) that is a 'Memory-read operation'.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x0051, L2D_CACHE_WR, Level 2 data cache access, write

If the [L2D_CACHE_RW](#) event is implemented, the counter counts each access counted by [L2D_CACHE_RW](#) that is a [Memory-write operation](#).

If the [L2D_CACHE_RW](#) event is not implemented, the counter counts each access counted by [L2D_CACHE](#) that is a [Memory-write operation](#).

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x0052, L2D_CACHE_REFILL_RD, Level 2 data cache refill, read

The counter counts each access counted by both [L2D_CACHE_RD](#) and [L2D_CACHE_REFILL](#).

That is, every refill of the Level 2 data or unified cache counted by [L2D_CACHE_REFILL](#) that is caused by a ‘Memory-read operation’.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x0053, [L2D_CACHE_REFILL_WR](#), Level 2 data cache refill, write

The counter counts each access counted by both [L2D_CACHE_REFILL](#) and [L2D_CACHE_WR](#).

That is, every refill of the Level 2 data or unified cache counted by [L2D_CACHE_REFILL](#) that is caused by a [Memory-write operation](#).

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x0056, [L2D_CACHE_WB_VICTIM](#), Level 2 data cache write-back, victim

The counter counts each write-back counted by [L2D_CACHE_WB](#) that occurs because of a capacity eviction due to a line being allocated into the cache..

It is IMPLEMENTATION DEFINED whether this includes capacity evictions due to an agent other the PE counting the event. For example, a stashing request from outside of the PE, or an allocation due to another PE that shares the cache.

In all cases, the event is counted only if the eviction is also an [L2D_CACHE_WB](#) event. This means that, if the cache is shared, the eviction is counted only if it is attributable to the PE counting the event, as defined by [L2D_CACHE_WB](#).

0x0057, [L2D_CACHE_WB_CLEAN](#), Level 2 data cache write-back, cleaning and coherency

The counter counts each write-back counted by [L2D_CACHE_WB](#) that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction..

It is IMPLEMENTATION DEFINED whether the transfer of a dirty cache line from the Level 1 data cache of this PE to the data cache of another PE due to a hardware coherency operation is counted when the dirty cache line is not also written back to a Level 2 cache or memory.

In all cases, the event is counted only if the write-back is also an [L2D_CACHE_WB](#) event. This means that, if the cache is shared, the write-back is counted only if it is attributable to the PE counting the event, as defined by [L2D_CACHE_WB](#).

0x0058, [L2D_CACHE_INVAL](#), Level 2 data cache invalidate

The counter counts each invalidation of a cache line in the Level 2 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 2 data or unified cache invalidates a line in the Level 2 data or unified cache.

The counter does not count locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 2 data cache is shared with other PEs. This event does not follow the general rule of Level 2 data cache events of only counting Attributable events.

0x005C, L2D_TLB_REFILL_RD, Level 2 data TLB refill, read

The counter counts each access counted by both [L2D_TLB_RD](#) and [L2D_TLB_REFILL](#).

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

0x005D, L2D_TLB_REFILL_WR, Level 2 data TLB refill, write

The counter counts each access counted by both [L2D_TLB_REFILL](#) and [L2D_TLB_WR](#).

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

0x005E, L2D_TLB_RD, Level 2 data TLB access, read

If the L2D_TLB_RW event is implemented, the counter counts each access counted by L2D_TLB_RW that is due to a demand 'Memory-read operation'.

If the L2D_TLB_RW event is not implemented, the counter counts each access counted by [L2D_TLB](#) that is due to a 'Memory-read operation'.

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

0x005F, L2D_TLB_WR, Level 2 data TLB access, write

If the L2D_TLB_RW event is implemented, the counter counts each access counted by L2D_TLB_RW that is due to a demand [Memory-write operation](#).

If the L2D_TLB_RW event is not implemented, the counter counts each access counted by [L2D_TLB](#) that is due to a [Memory-write operation](#).

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other

cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

0x0060, **BUS_ACCESS_RD, Bus access, read**

The counter counts each access counted by [BUS_ACCESS](#) that is a ‘Memory-read operation’.

0x0061, **BUS_ACCESS_WR, Bus access, write**

The counter counts each access counted by [BUS_ACCESS](#) that is a [Memory-write operation](#).

0x0062, **BUS_ACCESS_SHARED, Bus access, Normal, Cacheable, Shareable**

The counter counts each access counted by [BUS_ACCESS](#) that is Normal, Cacheable, Shareable.

Note

It is IMPLEMENTATION DEFINED how the PE translates the attributes from the translation table entry for a region to the attributes on the bus.

In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.

If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

0x0063, **BUS_ACCESS_NOT_SHARED, Bus access, not Normal, Cacheable, Shareable**

The counter counts each access counted by [BUS_ACCESS](#) that is not counted by [BUS_ACCESS_SHARED](#).

For example, the counter counts accesses marked as:

- Normal, Cacheable, Non-shareable.
- Normal, Non-cacheable.
- Device.

Note

It is IMPLEMENTATION DEFINED, how the PE translates the attributes from the translation table entries for a region to the attributes on the bus.

In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.

If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

0x0064, **BUS_ACCESS_NORMAL, Bus access, normal**

The counter counts each access counted by [BUS_ACCESS](#) that is to Normal or bulk memory.

For example, the counter counts Normal, Cacheable and Normal, Non-cacheable accesses but does not count Device accesses.

0x0065, **BUS_ACCESS_PERIPH, Bus access, peripheral**

The counter counts each access counted by [BUS_ACCESS](#) that is not counted by [BUS_ACCESS_NORMAL](#).

For example, the counter counts accesses to Device memory.

0x0066, **MEM_ACCESS_RD, Data memory access, read**

If the [MEM_ACCESS_RW](#) event is implemented, the counter counts each access counted by [MEM_ACCESS_RW](#) that is a ‘Memory-read operation’.

If the [MEM_ACCESS_RW](#) event is not implemented, the counter counts each access counted by [MEM_ACCESS](#) that is a ‘Memory-read operation’.

0x0067, **MEM_ACCESS_WR, Data memory access, write**

If the [MEM_ACCESS_RW](#) event is implemented, the counter counts each access counted by [MEM_ACCESS_RW](#) that is a [Memory-write operation](#).

If the [MEM_ACCESS_RW](#) event is not implemented, the counter counts each access counted by [MEM_ACCESS](#) that is a [Memory-write operation](#).

0x0068, **UNALIGNED_LD_SPEC, Unaligned access, read**

The counter counts each unaligned access counted by [UNALIGNED_LDST_SPEC](#) that is a ‘Memory-read operation’.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

0x0069, **UNALIGNED_ST_SPEC, Unaligned access, write**

The counter counts each unaligned access counted by [UNALIGNED_LDST_SPEC](#) that is a [Memory-write operation](#).

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

0x006A, **UNALIGNED_LDST_SPEC, Unaligned access**

The counter counts each access counted by [MEM_ACCESS](#) that is an unaligned ‘Memory-read operation’ or unaligned [Memory-write operation](#).

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

0x006C, **LDREX_SPEC, Exclusive operation speculatively executed, Load-Exclusive**

The counter counts each [speculatively executed](#) Load-Exclusive instruction.

The definition of [speculatively executed](#) is IMPLEMENTATION DEFINED but must be the same as for the [STREX_SPEC](#) event.

0x006D, **STREX_PASS_SPEC, Exclusive operation speculatively executed, Store-Exclusive pass**

The counter counts each [speculatively executed](#) Store-Exclusive instruction counted by [STREX_SPEC](#) that completed a write.

0x006E, **STREX_FAIL_SPEC, Exclusive operation speculatively executed, Store-Exclusive fail**

The counter counts each [speculatively executed](#) Store-Exclusive instruction counted by [STREX_SPEC](#) that fails to complete a write.

It is within the IMPLEMENTATION DEFINED definition of [speculatively executed](#) whether this includes conditional instructions that fail the condition code check.

0x006F, **STREX_SPEC, Exclusive operation speculatively executed, Store-Exclusive**

The counter counts each [speculatively executed](#) Store-Exclusive instruction.

The definition of [speculatively executed](#) is IMPLEMENTATION DEFINED but it must be the same as for the [LDREX_SPEC](#) event.

Arm recommends that this event is implemented if it is not possible to implement the exclusive operation [speculatively executed](#), Store-Exclusive pass, and exclusive operation [speculatively executed](#), Store-Exclusive fail, events with the same degree of speculation as the [LDREX_SPEC](#) event.

0x0070, **LD_SPEC, Operation speculatively executed, load**

The counter counts each operation counted by [LDST_SPEC](#) that is a load operation.

Operations due to 'Memory-reading instructions' are counted as load operations.

It is IMPLEMENTATION DEFINED whether operations due to the prefetch instructions counted by [PRF_SPEC](#) are counted as load operations:

- If operations due to the prefetch instructions are counted as load operations, then they are counted by [LD_SPEC](#).
- Otherwise, if operations due to the prefetch instructions are not counted as load operations by [LD_SPEC](#), then it is further IMPLEMENTATION DEFINED which one of the following applies:
 - They are counted as data-processing operations.
 - They are counted by [LDST_SPEC](#).

Arm recommends that if a prefetch operation is not implemented as a NOP and the [PRF_SPEC](#) event is not implemented, then the operation is counted as a load operation.

0x0071, **ST_SPEC, Operation speculatively executed, store**

The counter counts each operation counted by [LDST_SPEC](#) that is a store operation.

Operations due to 'Memory-writing instructions' are counted as store operations.

0x0072, **LDST_SPEC, Operation speculatively executed, load or store**

The counter counts each operation counted by [INST_SPEC](#) that is a load operation or a store operation.

See [LD_SPEC](#) and [ST_SPEC](#) for these classifications.

If the [PRF_SPEC](#) event is implemented and operations due to the prefetch instructions counted by [PRF_SPEC](#) are not counted as load operations by [LD_SPEC](#), then it is IMPLEMENTATION DEFINED whether these operations are counted by [LDST_SPEC](#). Arm recommends that these operations are counted by [LDST_SPEC](#).

0x0073, **DP_SPEC, Operation speculatively executed, integer data processing**

The counter counts each operation counted by [INST_SPEC](#) that is an integer data-processing operation.

An integer data-processing operation is any operation that is not counted as any of the following:

- A load or store operation, counted by [LDST_SPEC](#).
- A Software change of PC operation, counted by [PC_WRITE_SPEC](#).
- A scalar floating-point data processing operation, counted by [VFP_SPEC](#).
- An Advanced SIMD or SVE data processing operation, counted by [ASE_SVE_SPEC](#).
- A cryptographic operation, counted by [CRYPTO_SPEC](#).

That is, operations due to the following instructions are counted as integer data-processing operations:

- In AArch64 state instructions from the following sections:
 - Data processing - immediate.
 - Data processing - register.
 - System register instructions.

- ‘Instructions with register argument’.
- System instructions other than ‘Memory-writing instructions’.
- Hint instructions.
- In AArch32 state instructions from the following sections:
 - [Data-processing instructions](#).
 - [PSTATE and banked register access instructions](#).
 - [Banked register access instructions](#).
 - [Miscellaneous instructions](#) other than `ISB` and prefetches.
 - [System register access instructions](#) other than `LDC` and `STC` instructions.

0x0074, **ASE_SPEC, Operation speculatively executed, Advanced SIMD data processing**

The counter counts each operation counted by [INST_SPEC](#) that is an Advanced SIMD data-processing operation.

Operations due to the following instructions are counted as Advanced SIMD data-processing operations:

- For AArch64 state:
 - The SIMD operations listed in Data processing - SIMD and floating-point.
 - Advanced SIMD scalar instructions, including those which operate on floating-point values.
 - If [FEAT_Crypto](#) is implemented, the Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions. See [The Cryptographic Extension in AArch64 state](#).
- For AArch32 state:
 - Instructions from [Advanced SIMD data-processing instructions](#).
 - If [FEAT_Crypto](#) is implemented, the `VMULL` (P64 variant) instruction. See [The Cryptographic Extension in AArch32 state](#).

This includes all operations that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing operations.
- Scalar floating-point data-processing operations.
- Load or store operations.
- Cryptographic data-processing operations counted by [CRYPTO_SPEC](#). [CRYPTO_SPEC](#) does not count the Cryptographic data-processing operations included above.

0x0075, **VFP_SPEC, Operation speculatively executed, scalar floating-point data processing**

The counter counts each operation counted by [INST_SPEC](#) that is a scalar floating-point operation.

Operations due to the following instructions are counted as scalar floating-point data-processing operations:

- In AArch64 state:
 - The scalar floating-point operations from Data processing - SIMD and floating-point. Only the scalar operations are counted. SIMD floating-point operations are not counted.
 - Operations that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `FCVT{<mode>}`, `UCVTF`, and `SCVTF`.
- In AArch32 state:
 - Instructions from [Floating-point data-processing instructions](#).
 - Operations that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `VCVT<mode>` (floating-point), `VCVT`, `VCVTT`, and `VCVTB`.

This includes all scalar operations that operate on the SIMD&FP registers as floating-point values, except those that are counted as one of the following:

- SIMD scalar operations.
- Integer data-processing operations.
- Load or store operations.

0x0076, **PC_WRITE_SPEC**, Operation speculatively executed, Software change of the PC

The counter counts each operation counted by [INST_SPEC](#) that is a [Software change of the PC](#).

[Software change of the PC](#) operations are defined by the [PC_WRITE_RETIRED](#) event.

It is IMPLEMENTATION DEFINED whether the counter counts the operation when the branch is not taken.

0x0077, **CRYPTO_SPEC**, Operation speculatively executed, cryptographic data processing

The counter counts each operation counted by [INST_SPEC](#) that is a cryptographic operation.

Operations due to the following instructions are counted as cryptographic data-processing operations:

- In AArch64 state, instructions from The Cryptographic Extension in AArch64 state other than the include Advanced SIMD [PMULL](#) and [PMULL2](#) (1Q variants) instructions.
- In AArch32 state, instructions from [The Cryptographic Extension in AArch32 state](#) other than [VMULL](#) (P64 variant).

0x0078, **BR_IMMED_SPEC**, Branch speculatively executed, immediate branch

The counter counts each [speculatively executed](#) immediate branch instruction.

Operations due to the following instructions are counted as [speculatively executed](#) immediate branch instructions:

- For AArch32 state, the following instructions:
 - B{<c>} <label>.
 - BL{<c>} <label>.
 - BLX{<c>} <label>.
 - CBZ <Rn>, <label>.
 - CBNZ <label>.
- For AArch64 state, the following instructions:
 - B <label>.
 - B.cond <label>.
 - BL <label>.
 - CBZ <Rn>, <label>.
 - CBNZ <Rn>, <label>.
 - TBZ <Rn>, <label>.
 - TBNZ <Rn>, <label>.

Note

Conditional branches are always counted, regardless of whether the branch is taken or not taken.

If the Context synchronization barrier instruction ISB is counted as a [Software change of the PC](#) instruction by [PC_WRITE_SPEC](#), then it is IMPLEMENTATION DEFINED whether ISB is counted as a [speculatively executed](#) immediate branch instruction.

0x0079, **BR_RETURN_SPEC**, Branch speculatively executed, procedure return

The counter counts each [speculatively executed](#) procedure return instruction.

Operations due to the following instructions are counted as [speculatively executed](#) procedure return instructions:

- For AArch32 state, the following instructions:
 - BX R14.
 - MOV PC, LR.
 - POP {..., PC}.
 - LDR PC, [SP], #offset.
- For AArch64 state, the following instructions:
 - RET.

Note

The counter counts only the listed instructions as procedure returns. For example, it does not count the following AArch32 instructions as procedure return instructions:

- BX R0, because Rm != R14.
 - MOV PC, R0, because Rm != R14.
 - LDM SP, {..., PC}, because writeback is not specified.
 - LDR PC, [SP, #offset], because this specifies the wrong addressing mode.
-

0x007A, **BR_INDIRECT_SPEC, Branch speculatively executed, indirect branch**

The counter counts each [speculatively executed](#) indirect branch instruction.

Indirect branch instructions are [Software change of the PC](#) instructions other than exception-generating instructions and immediate branch instructions. [Software change of the PC](#) has the same definition as for the [PC_WRITE_SPEC](#) event. Immediate branch instructions are defined by the [BR_IMMED_SPEC](#) event.

0x007C, **ISB_SPEC, Barrier speculatively executed, ISB**

The counter counts each [speculatively executed](#) Instruction Synchronization Barrier instruction.

This includes the CP15ISB instruction.

0x007D, **DSB_SPEC, Barrier speculatively executed, DSB**

The counter counts each [speculatively executed](#) data synchronization and speculative load barrier instruction.

This includes the CP15DSB instruction.

0x007E, **DMB_SPEC, Barrier speculatively executed, DMB**

The counter counts each [speculatively executed](#) data memory barrier instruction.

This includes the CP15DSB instruction.

This does not include the implied barrier operations of load/store operations with release consistency semantics.

0x007F, **CSDB_SPEC, Barrier speculatively executed, CSDB**

The counter counts each [speculatively executed](#) control speculation barrier instruction.

0x0090, **RC_LD_SPEC, Release consistency operation speculatively executed, Load-Acquire**

The counter counts each [speculatively executed](#) ‘Memory-read operation’ with acquire or acquirepc semantics.

0x0091, **RC_ST_SPEC, Release consistency operation speculatively executed, Store-Release**

The counter counts each [speculatively executed Memory-write operation](#) with release semantics.

0x00A0, **L3D_CACHE_RD, Level 3 data cache access, read**

If the [L3D_CACHE_RW](#) event is implemented, the counter counts each access counted by [L3D_CACHE_RW](#) that is a ‘Memory-read operation’.

If the [L3D_CACHE_RW](#) event is not implemented, the counter counts each access counted by [L3D_CACHE](#) that is a ‘Memory-read operation’.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is

shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x00A1, **L3D_CACHE_WR, Level 3 data cache access, write**

If the `L3D_CACHE_RW` event is implemented, the counter counts each access counted by `L3D_CACHE_RW` that is a Memory-write operation.

If the `L3D_CACHE_RW` event is not implemented, the counter counts each access counted by `L3D_CACHE` that is a Memory-write operation.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x00A2, **L3D_CACHE_REFILL_RD, Level 3 data cache refill, read**

The counter counts each access counted by both `L3D_CACHE_RD` and `L3D_CACHE_REFILL`.

That is, every refill of the Level 3 data or unified cache counted by `L3D_CACHE_REFILL` that is caused by a 'Memory-read operation'.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x00A3, **L3D_CACHE_REFILL_WR, Level 3 data cache refill, write**

The counter counts each access counted by both `L3D_CACHE_REFILL` and `L3D_CACHE_WR`.

That is, every refill of the Level 3 data or unified cache counted by `L3D_CACHE_REFILL` that is caused by a Memory-write operation.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

0x00A6, **L3D_CACHE_WB_VICTIM, Level 3 data cache write-back, victim**

The counter counts each write-back counted by `L3D_CACHE_WB` that occurs because of a capacity eviction due to a line being allocated into the cache..

It is IMPLEMENTATION DEFINED whether this includes capacity evictions due to an agent other the PE counting the event. For example, a stashing request from outside of the PE, or an allocation due to another PE that shares the cache.

In all cases, the event is counted only if the eviction is also an `L3D_CACHE_WB` event. This means that, if the cache is shared, the eviction is counted only if it is attributable to the PE counting the event, as defined by `L3D_CACHE_WB`.

0x00A7, **L3D_CACHE_WB_CLEAN, Level 3 data cache write-back, cleaning and coherency**

The counter counts each write-back counted by [L3D_CACHE_WB](#) that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction..

It is IMPLEMENTATION DEFINED whether the transfer of a dirty cache line from the Level 1 data cache of this PE to the data cache of another PE due to a hardware coherency operation is counted when the dirty cache line is not also written back to a Level 2 cache or memory.

In all cases, the event is counted only if the write-back is also an [L3D_CACHE_WB](#) event. This means that, if the cache is shared, the write-back is counted only if it is attributable to the PE counting the event, as defined by [L3D_CACHE_WB](#).

0x00A8, **L3D_CACHE_INVALID, Level 3 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 3 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 3 data or unified cache invalidates a line in the Level 3 data or unified cache.

The counter does not count locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 3 data cache is shared with other PEs. This event does not follow the general rule of Level 3 data cache events of only counting Attributable events.

ASE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, Advanced SIMD add or subtract

The counter counts each [speculatively executed](#) floating-point add or subtract operation counted by [ASE_SVE_FP_ADDSUB_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FABD, FADD, or FSUB.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

ASE_FP_BF16_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each Advanced SIMD data-processing operation counted by [ASE_SVE_FP_BF16_MIN_SPEC](#) due to an instruction where the smallest type was BFloat16 floating-point.

ASE_FP_CVT_SPEC, Floating-point operation speculatively executed, Advanced SIMD convert

The counter counts each [speculatively executed](#) floating-point convert operation counted by [ASE_SVE_FP_CVT_SPEC](#) due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point convert operations are counted in AArch32 state.

ASE_FP_DIV_SPEC, Floating-point operation speculatively executed, Advanced SIMD divide

The counter counts each [speculatively executed](#) floating-point divide operation counted by [ASE_SVE_FP_DIV_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FDIV.

It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.

ASE_FP_DOT_SPEC, Floating-point operation speculatively executed, Advanced SIMD dot-product

The counter counts each dot-product operation counted by [ASE_SVE_FP_DOT_SPEC](#) due to any of the following A64 instructions:

It is IMPLEMENTATION DEFINED which dot-product operations are counted in AArch32 state.

ASE_FP_DP_SPEC, Floating-point operation speculatively executed, Advanced SIMD double-precision, data-processing

The counter counts each [speculatively executed](#) double-precision floating-point data-processing operation counted by [ASE_SVE_FP_DP_SPEC](#) due to an Advanced SIMD instruction.

ASE_FP_FMA_SPEC, Floating-point operation speculatively executed, Advanced SIMD FMA

The counter counts each [speculatively executed](#) floating-point fused multiply-add or multiply-subtract operation counted by [ASE_SVE_FP_FMA_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FMLA or FMLS.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

ASE_FP_FP8_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each Advanced SIMD data-processing operation counted by [ASE_SVE_FP_FP8_MIN_SPEC](#) due to an instruction where the smallest type was 8-bit floating-point.

ASE_FP_HP_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each Advanced SIMD data-processing operation counted by [ASE_SVE_FP_HP_MIN_SPEC](#) due to an instruction where the smallest type was half-precision floating-point.

ASE_FP_HP_SPEC, Floating-point operation speculatively executed, Advanced SIMD half-precision, data-processing

The counter counts each [speculatively executed](#) half-precision floating-point data-processing operation counted by [ASE_SVE_FP_HP_SPEC](#) due to an Advanced SIMD instruction.

ASE_FP_MMLA_SPEC, Floating-point operation speculatively executed, Advanced SIMD matrix multiply

The counter counts each floating-point matrix multiply operation counted by [ASE_SVE_FP_MMLA_SPEC](#) due to any of the following A64 instructions:

It is IMPLEMENTATION DEFINED which floating-point matrix multiply operations are counted in AArch32 state.

ASE_FP_MUL_SPEC, Floating-point operation speculatively executed, Advanced SIMD multiply

The counter counts each [speculatively executed](#) floating-point multiply operation counted by [ASE_SVE_FP_MUL_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FMUL or FMULX.

It is IMPLEMENTATION DEFINED which floating-point multiply operations are counted in AArch32 state.

ASE_FP_PREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD pairwise add step or pairwise reduce step

The counter counts each [speculatively executed](#) floating-point pairwise add step or pairwise reduce step operation counted by [ASE_FP_VREDUCE_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXP, FMINNMP, or FMINP.

It is IMPLEMENTATION DEFINED which floating-point pairwise add step or pairwise reduce step operations are counted in AArch32 state.

ASE_FP_RECPE_SPEC, Floating-point operation speculatively executed, Advanced SIMD reciprocal estimate

The counter counts each [speculatively executed](#) floating-point reciprocal estimate operation counted by [ASE_SVE_FP_RECPE_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FRECPPE or FRSQRTE.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.

ASE_FP_SPEC, Floating-point operation speculatively executed, Advanced SIMD

The counter counts each [speculatively executed](#) floating-point operation counted by [ASE_SVE_FP_SPEC](#) due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point operations are counted in AArch32 state.

ASE_FP_SP_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each Advanced SIMD data-processing operation counted by [ASE_SVE_FP_SP_MIN_SPEC](#) due to an instruction where the smallest type was single-precision floating-point.

ASE_FP_SP_SPEC, Floating-point operation speculatively executed, Advanced SIMD single-precision, data-processing

The counter counts each [speculatively executed](#) single-precision floating-point data-processing operation counted by [ASE_SVE_FP_SP_SPEC](#) due to an Advanced SIMD instruction.

ASE_FP_SQRT_SPEC, Floating-point operation speculatively executed, Advanced SIMD square root

The counter counts each [speculatively executed](#) floating-point square-root operation counted by [ASE_SVE_FP_SQRT_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FSQRT.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

ASE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD pairwise or reduction

The counter counts each [speculatively executed](#) floating-point pairwise reduction or pairwise operation counted by [ASE_SVE_FP_VREDUCE_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXNMV, FMAXP, FMAXV, FMINNMP, FMINNMV, FMINP, or FMINV.

Note

Treewise reduction instructions might be executed and counted as multiple pairwise operations.

ASE_INST_SPEC, Operation speculatively executed, Advanced SIMD

The counter counts each [speculatively executed](#) operation counted by [ASE_SVE_INST_SPEC](#) that is classified as an Advanced SIMD operation.

Operations due to the following instructions are counted as Advanced SIMD operations:

- For AArch64 state:
 - The SIMD operations listed in Data processing - SIMD and floating-point.
 - If [FEAT_Crypto](#) is implemented, the Advanced SIMD Cryptographic instructions. See The Cryptographic Extension in AArch64 state.
 - Advanced SIMD loads and stores.
- For AArch32 state:
 - Instructions from [Advanced SIMD data-processing instructions](#).
 - If [FEAT_Crypto](#) is implemented, the `VMULL` (P64 variant) instruction. See [The Cryptographic Extension in AArch32 state](#).

This includes all operations that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing operations.
- Scalar floating-point data-processing operations.

It is IMPLEMENTATION DEFINED whether Advanced SIMD scalar operations are counted as Advanced SIMD operations. This includes Advanced SIMD scalar instructions which operate on floating-point values.

ASE_INT16_SPEC, Integer operation speculatively executed, Advanced SIMD 16-bit

The counter counts each operation counted by [ASE_SVE_INT16_SPEC](#) that is an Advanced SIMD operation.

ASE_INT32_SPEC, Integer operation speculatively executed, Advanced SIMD 32-bit

The counter counts each operation counted by [ASE_SVE_INT32_SPEC](#) that is an Advanced SIMD operation.

ASE_INT64_SPEC, Integer operation speculatively executed, Advanced SIMD 64-bit

The counter counts each operation counted by [ASE_SVE_INT64_SPEC](#) that is an Advanced SIMD operation.

ASE_INT8_SPEC, Integer operation speculatively executed, Advanced SIMD 8-bit

The counter counts each operation counted by [ASE_SVE_INT8_SPEC](#) that is an Advanced SIMD operation.

ASE_INT_DOT_SPEC, Operation speculatively executed, Advanced SIMD integer dot-product

The counter counts each integer dot product operation counted by [ASE_SVE_INT_DOT_SPEC](#) due to any of the following A64 instructions:

It is IMPLEMENTATION DEFINED which integer dot product operations are counted in AArch32 state.

ASE_INT_MMLA_SPEC, Integer operation speculatively executed, Advanced SIMD matrix multiply

The counter counts each integer matrix multiply operation counted by [ASE_SVE_INT_MMLA_SPEC](#) due to any of the following A64 instructions:

It is IMPLEMENTATION DEFINED which integer matrix multiply operations are counted in AArch32 state.

ASE_INT_MUL_SPEC, Integer operation speculatively executed, Advanced SIMD multiply

The counter counts each [speculatively executed](#) integer multiply or multiply-accumulate operation counted by [ASE_SVE_INT_MUL_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

ASE_INT_SPEC, Integer operation speculatively executed, Advanced SIMD

The counter counts each [speculatively executed](#) integer operation counted by [ASE_SVE_INT_SPEC](#) due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD integer operations are counted in AArch32 state.

ASE_INT_VREDUCE_SPEC, Integer operation speculatively executed, Advanced SIMD reduction

The counter counts each [speculatively executed](#) across-vector and pairwise integer reduction operation counted by [ASE_SVE_INT_VREDUCE_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: ADDP, ADDV, SADALP, SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADALP, UADDLP, UADDLV, UMAXP, UMAXV, UMINP, or UMINV.

It is IMPLEMENTATION DEFINED which across-vector and pairwise integer reduction operations are counted in AArch32 state.

ASE_LUT_SPEC, Lookup table operation speculatively executed, Advanced SIMD data processing

The counter counts each [speculatively executed](#) LUT operation counted by [SE_LUT_SPEC](#) that returns a value from a lookup table made up of a V register which is indexed by values from a V register.

ASE_NONFP_SPEC, Non-floating-point operation speculatively executed, Advanced SIMD

The counter counts each [speculatively executed](#) non-floating-point operation counted by [ASE_SVE_NONFP_SPEC](#) due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD non-floating-point data-processing operations are counted in AArch32 state.

ASE_SVE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE add or subtract

The counter counts each [speculatively executed](#) floating-point add or subtract operation counted by [FP_ADDSUB_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FABD, FADD, or FSUB.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

ASE_SVE_FP_BF16_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each Advanced SIMD or SVE data-processing operation counted by [SE_FP_BF16_MIN_SPEC](#) due to an instruction where the smallest type was BFloat16 floating-point.

ASE_SVE_FP_BF16_SPEC, Floating-point operation speculatively executed, SVE or Advanced SIMD BFloat16, data-processing

The counter counts each [speculatively executed](#) BFloat16 floating-point data-processing operation counted by FP_BF16_SPEC due to an Advanced SIMD.

ASE_SVE_FP_CVT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE convert

The counter counts each [speculatively executed](#) floating-point convert operation counted by FP_CVT_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point convert operations are counted in AArch32 state.

ASE_SVE_FP_DIV_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE divide

The counter counts each [speculatively executed](#) floating-point divide operation counted by FP_DIV_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FDIV.

It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.

ASE_SVE_FP_DOT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE dot-product

The counter counts each dot-product operation counted by FP_SPEC due to any of the following A64 instructions:

It is IMPLEMENTATION DEFINED which dot-product operations are counted in AArch32 state.

ASE_SVE_FP_DP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE double-precision, data-processing

The counter counts each [speculatively executed](#) double-precision floating-point data-processing operation due to an Advanced SIMD.

ASE_SVE_FP_FMA_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE FMA

The counter counts each [speculatively executed](#) floating-point fused multiply-add or multiply-subtract operation counted by FP_FMA_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMLA or FMLS.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

ASE_SVE_FP_FP8_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

ASE_SVE_FP_HP_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

ASE_SVE_FP_HP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE half-precision

The counter counts each [speculatively executed](#) half-precision floating-point operation counted by [FP_HP_SPEC](#) due to an Advanced SIMD.

ASE_SVE_FP_MMLA_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE matrix multiply

The counter counts each floating-point matrix multiply operation counted by [FP_SPEC](#) due to any of the following A64 instructions:

It is IMPLEMENTATION DEFINED which floating-point matrix multiply operations are counted in AArch32 state.

ASE_SVE_FP_MUL_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE multiply

The counter counts each [speculatively executed](#) floating-point fused multiply-add or multiply-subtract operation counted by [FP_MUL_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FMUL or FMULX.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

ASE_SVE_FP_RECPE_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE reciprocal estimate

The counter counts each [speculatively executed](#) floating-point reciprocal estimate operation counted by [FP_RECPE_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRTE.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.

ASE_SVE_FP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE

The counter counts each [speculatively executed](#) floating-point operation counted by [SE_FP_SPEC](#) due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD floating-point operations are counted in AArch32 state.

ASE_SVE_FP_SP_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each Advanced SIMD or SVE data-processing operation counted by [SE_FP_SP_MIN_SPEC](#) due to an instruction where the smallest type was single-precision floating-point.

ASE_SVE_FP_SP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE single-precision, data-processing

The counter counts each [speculatively executed](#) single-precision floating-point data-processing operation counted by [FP_SP_SPEC](#) due to an Advanced SIMD.

ASE_SVE_FP_SQRT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE square-root

The counter counts each [speculatively executed](#) floating-point square-root operation counted by [FP_SQRT_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FSQRT.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

ASE_SVE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE vector reduction

The counter counts each [speculatively executed](#) floating-point reduction operation counted by [ASE_SVE_FP_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXNMV, FMAXP, FMAXV, FMINNMP, FMINNMPV, FMINP, or FMINV.

It is IMPLEMENTATION DEFINED which floating-point reduction operations are counted in AArch32 state.

ASE_SVE_INST_SPEC, Operation speculatively executed, Advanced SIMD or SVE

The counter counts each [speculatively executed](#) operation counted by [SE_INST_SPEC](#) that is classified as an Advanced SIMD operation or an SVE operation.

See [ASE_INST_SPEC](#) and [SVE_INST_SPEC](#) for these classifications.

ASE_SVE_INT16_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 16-bit

The counter counts each operation counted by [ASE_SVE_INT_SPEC](#) where the largest type is 16-bit integer.

ASE_SVE_INT32_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 32-bit

The counter counts each operation counted by [ASE_SVE_INT_SPEC](#) where the largest type is 32-bit integer.

ASE_SVE_INT64_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 64-bit

The counter counts each operation counted by [ASE_SVE_INT_SPEC](#) where the largest type is 64-bit integer.

ASE_SVE_INT8_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 8-bit

The counter counts each operation counted by [ASE_SVE_INT_SPEC](#) where the largest type is 8-bit integer.

ASE_SVE_INT_DOT_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE dot-product

The counter counts each integer dot product operation counted by [INT_SPEC](#) due to any of the following A64 instructions:

It is IMPLEMENTATION DEFINED which integer dot product operations are counted in AArch32 state.

ASE_SVE_INT_MMLA_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE matrix multiply

The counter counts each integer matrix multiply operation counted by [INT_SPEC](#) due to any of the following A64 instructions:

It is IMPLEMENTATION DEFINED which integer matrix multiply operations are counted in AArch32 state.

ASE_SVE_INT_MUL_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE multiply

The counter counts each [speculatively executed](#) integer multiply or multiply-accumulate operation counted by [INT_MUL_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

ASE_SVE_INT_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE

The counter counts each [speculatively executed](#) integer arithmetic operation counted by [SE_INT_SPEC](#) due to an A64 Advanced SIMD.

It is IMPLEMENTATION DEFINED which Advanced SIMD integer operations are counted in AArch32 state.

ASE_SVE_INT_VREDUCE_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE reduction

The counter counts each [speculatively executed](#) across-vector and pairwise integer reduction operation counted by [INT_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: ADDP, ADDV, SADALP, SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADALP, UADDLP, UADDLV, UMAXP, UMAXV, UMINP, or UMINV.

It is IMPLEMENTATION DEFINED which across-vector and pairwise integer reduction operations are counted in AArch32 state.

ASE_SVE_LDST_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous load or store multiple vector

The counter counts each [speculatively executed](#) operation due to an A64 Advanced SIMD multiple vector contiguous structure load or store instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure load or store operations are counted in AArch32 state.

ASE_SVE_LDST_SPEC, Operation speculatively executed, Advanced SIMD or SVE load or store

The counter counts each [speculatively executed](#) operation that reads from or writes to memory due to an A64 Advanced SIMD load, store, or prefetch instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD load or store operations are counted in AArch32 state.

ASE_SVE_LD_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous load multiple vector

The counter counts each [speculatively executed](#) operation that reads from memory counted by [ASE_SVE_LDST_MULTI_SPEC](#) due to an A64 Advanced SIMD multiple vector contiguous structure load instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure load operations are counted in AArch32 state.

ASE_SVE_LD_SPEC, Operation speculatively executed, Advanced SIMD or SVE load

The counter counts each [speculatively executed](#) operation that reads from memory counted by [ASE_SVE_LDST_SPEC](#) due to an A64 Advanced SIMD load instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD load operations are counted in AArch32 state.

ASE_SVE_NONFP_SPEC, Non-floating-point operation speculatively executed, Advanced SIMD or SVE

The counter counts each [speculatively executed](#) non-floating-point operation counted by [NONFP_SPEC](#) due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD non-floating-point data-processing operations are counted in AArch32 state.

ASE_SVE_SPEC, Operation speculatively executed, Advanced SIMD data processing or SVE data processing

The counter counts each operation counted by [INST_SPEC](#) that is an Advanced SIMD data-processing operation or an SVE data-processing operation.

See [ASE_SPEC](#) and [SVE_SPEC](#) for these classifications.

ASE_SVE_ST_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous store multiple vector

The counter counts each [speculatively executed](#) operation that writes to memory counted by [ASE_SVE_LDST_MULTI_SPEC](#) due to an A64 Advanced SIMD multiple vector contiguous structure store instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD contiguous structure store operations are counted in AArch32 state.

ASE_SVE_ST_SPEC, Operation speculatively executed, Advanced SIMD or SVE store

The counter counts each [speculatively executed](#) operation that writes to memory counted by [ASE_SVE_LDST_SPEC](#) due to an A64 Advanced SIMD store instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD store operations are counted in AArch32 state.

ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC, Advanced SIMD or SVE unaligned contiguous access

The counter counts each [speculatively executed](#) Advanced SIMD contiguous load or store operation where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD load or store operations are counted in AArch32 state.

ASE_SVE_UNALIGNED_CONTIG_LD_SPEC, Advanced SIMD or SVE unaligned contiguous read

The counter counts each unaligned memory access counted by [ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC](#) due to an A64 Advanced SIMD load instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD load operations are counted in AArch32 state.

ASE_SVE_UNALIGNED_CONTIG_ST_SPEC, Advanced SIMD or SVE unaligned contiguous write

The counter counts each unaligned memory access counted by [ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC](#) due to a A64 Advanced SIMD store instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned contiguous Advanced SIMD store operations are counted in AArch32 state.

ASE_SVE_UNALIGNED_LDST_SPEC, Advanced SIMD or SVE unaligned access

The counter counts each [speculatively executed](#) A64 Advanced SIMD load or store instruction where either:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather, scatter, or single element address is not aligned to the memory element access size, in bytes.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD load or store operations are counted in AArch32 state.

ASE_SVE_UNALIGNED_LD_SPEC, Advanced SIMD or SVE unaligned read

The counter counts each unaligned memory access counted by [ASE_SVE_UNALIGNED_LDST_SPEC](#) due to an A64 Advanced SIMD load instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD load operations are counted in AArch32 state.

ASE_SVE_UNALIGNED_ST_SPEC, Advanced SIMD or SVE unaligned write

The counter counts each unaligned memory access counted by [ASE_SVE_UNALIGNED_LDST_SPEC](#) due to an A64 Advanced SIMD store instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

It is IMPLEMENTATION DEFINED which unaligned Advanced SIMD store operations are counted in AArch32 state.

ASE_SVE_UOP_SPEC, Microarchitectural operation speculatively executed, Advanced SIMD or SVE

The counter counts each [speculatively executed](#) microarchitectural operation counted by [UOP_SPEC](#) due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitectural operations due to Advanced SIMD scalar instructions.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

ASE_UOP_SPEC, Microarchitectural operation speculatively executed, Advanced SIMD

The counter counts each [speculatively executed](#) microarchitectural operation counted by [ASE_SVE_UOP_SPEC](#) due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitectural operations due to Advanced SIMD scalar instructions.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

BASE_LDST_REG_SPEC, Operation speculatively executed, general-purpose register load, store, or prefetch

The counter counts each [speculatively executed](#) operation that reads from or writes to memory due to a general-purpose register load, store, or prefetch instruction.

It is IMPLEMENTATION DEFINED which load, store, and prefetch operations are counted in AArch32 state.

BASE_LD_REG_SPEC, Operation speculatively executed, general-purpose register load

The counter counts each [speculatively executed](#) operation that reads from memory counted by [BASE_LDST_REG_SPEC](#) due to a general-purpose register load instruction.

It is IMPLEMENTATION DEFINED which load operations are counted in AArch32 state.

BASE_PRF_SPEC, Operation speculatively executed, general-purpose register prefetch

The counter counts each [speculatively executed](#) prefetch operation counted by [BASE_LDST_REG_SPEC](#) due to any of the following A64 instructions:

- Scalar: PRFM, PRFUM, or RPRFM.

It is IMPLEMENTATION DEFINED which prefetch operations are counted in AArch32 state.

BASE_ST_REG_SPEC, Operation speculatively executed, general-purpose register store

The counter counts each [speculatively executed](#) operation that writes to memory counted by [BASE_LDST_REG_SPEC](#) due to a general-purpose register store instruction.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to DC ZVA instructions.

It is IMPLEMENTATION DEFINED which store operations are counted in AArch32 state.

BR_IMMED_MIS_PRED_RETIRE, Branch instruction architecturally executed, mispredicted immediate

The counter counts each instruction counted by both [BR_IMMED_RETIRE](#) and [BR_MIS_PRED_RETIRE](#).

These are all direct branch instructions on the architecturally executed path, where the branch is mispredicted.

BR_IMMED_PRED_RETIRE, Branch instruction architecturally executed, predicted immediate

The counter counts each instruction counted by both [BR_IMMED_RETIRE](#) and [BR_PRED_RETIRE](#).

These are all direct branch instructions on the architecturally executed path, where the branch is correctly predicted.

BR_INDNR_MIS_PRED_RETIRE, Branch instruction architecturally executed, mispredicted indirect excluding procedure return

The counter counts each instruction counted by [BR_IND_MIS_PRED_RETIRE](#) where if taken, the branch would not be counted by [BR_RETURN_RETIRE](#).

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is mispredicted.

BR_INDNR_PRED_RETIRE, Branch instruction architecturally executed, predicted indirect excluding procedure return

The counter counts each instruction counted by [BR_IND_PRED_RETIRE](#) where if taken, the branch would not be counted by [BR_RETURN_RETIRE](#).

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is correctly predicted.

BR_IND_MIS_PRED_RETIRE, Branch instruction architecturally executed, mispredicted indirect

The counter counts each instruction counted by both [BR_IND_RETIRE](#) and [BR_MIS_PRED_RETIRE](#).

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is mispredicted.

BR_IND_PRED_RETIRE, Branch instruction architecturally executed, predicted indirect

The counter counts each instruction counted by both [BR_IND_RETIRE](#) and [BR_PRED_RETIRE](#).

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is correctly predicted.

BR_PRED_RETIRE, Branch instruction architecturally executed, predicted branch

The counter counts each instruction counted by [BR_RETIRE](#) that is not counted by [BR_MIS_PRED_RETIRE](#).

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted.

BR_RETURN_MIS_PRED_RETIRE, Branch instruction architecturally executed, mispredicted procedure return

The counter counts each instruction counted by [BR_IND_MIS_PRED_RETIRE](#) where if taken, the branch would also be counted by [BR_RETURN_RETIRE](#).

These are all return instructions on the architecturally executed path, where the branch is mispredicted.

BR_RETURN_PRED_RETIRE, Branch instruction architecturally executed, predicted procedure return

The counter counts each instruction counted by [BR_IND_PRED_RETIRE](#) where if taken, the branch would be counted by [BR_RETURN_RETIRE](#).

These are all return instructions on the architecturally executed path, where the branch is correctly predicted.

BR_SKIP_MIS_PRED_RETIRE, Branch instruction architecturally executed, mispredicted branch, not taken

The counter counts each instruction counted by both [BR_MIS_PRED_RETIRE](#) and [BR_SKIP_RETIRE](#).

These are all branch instructions on the architecturally executed path, where the branch is mispredicted and not taken.

BR_SKIP_PRED_RETIRE, Branch instruction architecturally executed, predicted branch, not taken

The counter counts each instruction counted by both [BR_PRED_RETIRE](#) and [BR_SKIP_RETIRE](#).

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted and not taken.

BR_TAKEN_MIS_PRED_RETIRE, Branch instruction architecturally executed, mispredicted branch, taken

The counter counts each instruction counted by both [BR_MIS_PRED_RETIRE](#) and [PC_WRITE_RETIRE](#).

These are all branch instructions on the architecturally executed path, where the branch is mispredicted and taken.

BR_TAKEN_PRED_RETIRE, Branch instruction architecturally executed, predicted branch, taken

The counter counts each instruction counted by both [BR_PRED_RETIRE](#) and [PC_WRITE_RETIRE](#).

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted and taken.

BUS_REQ, Bus request

The counter counts each request generated by a 'Memory-read operation' or [Memory-write operation](#) that accesses outside of the boundary of the PE and its closely-coupled caches.

Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. Where an implementation has multiple buses at this boundary, this event counts the sum of requests across all buses.

A bus request is the start of a bus transaction. The exact nature of a bus transaction is IMPLEMENTATION DEFINED, but for the purposes of event monitoring consists of a single access comprising one or more cycles, or beats, when the transaction occupies the bus. The [BUS_REQ](#) event therefore counts the number of transactions. The [BUS_ACCESS](#) event counts the occupancy of the transaction on the bus.

Bus transactions include refills of and write-backs from data, instruction, and unified caches. Whether bus transactions include operations that use the bus but do not explicitly transfer data is IMPLEMENTATION DEFINED. This must be the same for the [BUS_REQ](#) and [BUS_ACCESS](#) events.

An Unattributable bus transaction occurs when a requester outside the PE makes a request that results in a bus transaction, for example, a coherency request.

The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

If the bus is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

BUS_REQ_RD, Bus request, read

The counter counts each transaction counted by `BUS_REQ` that is a ‘Memory-read operation’.

BUS_REQ_RD_PERCYC, Bus read transactions in progress

The counter increments by the number of bus read transactions counted by `BUS_REQ_RD` in progress on each `Processor cycle`.

The ratio $\text{BUS_REQ_RD_PERCYC} \div \text{BUS_REQ_RD}$ is the mean duration of bus read transactions in `Processor cycles`.

A bus read transaction is in progress between the transaction starting and it completing. This includes cycles when the transaction is not occupying the bus.

If the bus is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

BUS_REQ_WR, Bus request, write

The counter counts each transaction counted by `BUS_REQ` that is a `Memory-write operation`.

BUS_REQ_WR_PERCYC, Bus write transactions in progress

The counter increments by the number of bus write transactions counted by `BUS_REQ_WR` in progress on each `Processor cycle`.

The ratio $\text{BUS_REQ_WR_PERCYC} \div \text{BUS_REQ_WR}$ is the mean duration of bus write transactions in `Processor cycles`.

A bus write transaction is in progress between the transaction starting and it completing. This includes cycles when the transaction is not occupying the bus.

A mean duration value calculated using this event only measures the latency from the perspective of the bus. This is not necessarily the same as the latency for the write to be Complete, as the write might be completed on the bus before reaching its endpoint.

For some bus implementations, writes are posted, meaning that, from the perspective of the bus, the write completes immediately. There is no response when the write reaches its Completion endpoint. This event is not included for such implementations.

If the bus is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

CNT_CYCLES, Constant frequency cycles

The counter increments at a constant frequency when the PE is not in `WFI` or `WFE` state, equal to the rate of increment of the System Counter, `CNTPCT_EL0`.

When the PE is in `WFI` or `WFE` state, it is IMPLEMENTATION SPECIFIC whether the counter counts.

`PMCEID0_EL0[36]` reads as 1 if this event is implemented and 0 otherwise.

CTI_TRIGOUT4, Cross-trigger Interface output trigger 4

The counter counts each event signaled on CTI output trigger 4.

Note

CTI output triggers are input events to the PMU and trace unit.

`PMCEID0_EL0[56]` reads as 1 if this event is implemented and 0 otherwise.

CTI_TRIGOUT5, Cross-trigger Interface output trigger 5

The counter counts each event signaled on CTI output trigger 5.

Note

CTI output triggers are input events to the PMU and trace unit.

`PMCEID0_EL0[57]` reads as 1 if this event is implemented and 0 otherwise.

CTI_TRIGOUT6, Cross-trigger Interface output trigger 6

The counter counts each event signaled on CTI output trigger 6.

Note

CTI output triggers are input events to the PMU and trace unit.

`PMCEID0_EL0[58]` reads as 1 if this event is implemented and 0 otherwise.

CTI_TRIGOUT7, Cross-trigger Interface output trigger 7

The counter counts each event signaled on CTI output trigger 7.

Note

CTI output triggers are input events to the PMU and trace unit.

`PMCEID0_EL0[59]` reads as 1 if this event is implemented and 0 otherwise.

DSNP_HIT, Snoop hit, data

The counter counts each snoop that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HITM_FAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in far cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by `DSNP_HITM_LDST_RETIRED` that hit modified data outside the local PE cluster on the same device.

DSNP_HITM_FAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in far cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HITM_FAR_LDST_RETIRED](#) that is a Memory-reading instruction.

DSNP_HITM_FAR_RD, Snoop hit in far local cache, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_FAR_RD](#) that accesses a previously modified cache location.

DSNP_HITM_FAR_RW, Snoop hit in far local cache, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_FAR_RW](#) that accesses a previously modified cache location.

DSNP_HITM_FAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in far cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HITM_FAR_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

DSNP_HITM_FAR_WR, Snoop hit in far local cache, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_FAR_WR](#) that accesses a previously modified cache location.

DSNP_HITM_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that hit modified data in a data cache outside the cache hierarchy of this PE.

DSNP_HITM_LD_RETIRED, Load instruction architecturally executed, snoop hit

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HITM_LDST_RETIRED](#) that is a Memory-reading instruction.

DSNP_HITM_N1_RD, Snoop hit in cache at distance 1, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_N1_RD](#) that accesses a previously modified cache location.

DSNP_HITM_N1_RW, Snoop hit in cache at distance 1, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_N1_RW](#) that accesses a previously modified cache location.

DSNP_HITM_N1_WR, Snoop hit in cache at distance 1, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_N1_WR](#) that accesses a previously modified cache location.

DSNP_HITM_N2_RD, Snoop hit in cache at distance 2, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_N2_RD](#) that accesses a previously modified cache location.

DSNP_HITM_N2_RW, Snoop hit in cache at distance 2, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_N2_RW](#) that accesses a previously modified cache location.

DSNP_HITM_N2_WR, Snoop hit in cache at distance 2, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_N2_WR](#) that accesses a previously modified cache location.

DSNP_HITM_N3_RD, Snoop hit in cache at distance 3, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_N3_RD](#) that accesses a previously modified cache location.

DSNP_HITM_N3_RW, Snoop hit in cache at distance 3, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_N3_RW](#) that accesses a previously modified cache location.

DSNP_HITM_N3_WR, Snoop hit in cache at distance 3, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_N3_WR](#) that accesses a previously modified cache location.

DSNP_HITM_N4_RD, Snoop hit in cache at distance 4, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_N4_RD](#) that accesses a previously modified cache location.

DSNP_HITM_N4_RW, Snoop hit in cache at distance 4, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_N4_RW](#) that accesses a previously modified cache location.

DSNP_HITM_N4_WR, Snoop hit in cache at distance 4, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_N4_WR](#) that accesses a previously modified cache location.

DSNP_HITM_NEAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in near cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [DSNP_HITM_NEAR_LDST_RETIRED](#) that hit modified data outside of the cache hierarchy of this PE in the local PE cluster.

DSNP_HITM_NEAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in near cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HITM_NEAR_LDST_RETIRED](#) that is a Memory-reading instruction.

DSNP_HITM_NEAR_RD, Snoop hit in near local cache, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_NEAR_RD](#) that accesses a previously modified cache location.

DSNP_HITM_NEAR_RW, Snoop hit in near local cache, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_NEAR_RW](#) that accesses a previously modified cache location.

DSNP_HITM_NEAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in near cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HITM_NEAR_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

DSNP_HITM_NEAR_WR, Snoop hit in near local cache, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_NEAR_WR](#) that accesses a previously modified cache location.

DSNP_HITM_RD, Snoop hit, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_RD](#) that accesses a previously modified cache location.

DSNP_HITM_REMOTE_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in remote cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [DSNP_HITM_LDST_RETIRED](#) that hit modified data on a remote device.

DSNP_HITM_REMOTE_LD_RETIRED, Load instruction architecturally executed, snoop hit in remote cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HITM_REMOTE_LDST_RETIRED](#) that is a Memory-reading instruction.

DSNP_HITM_REMOTE_RD, Snoop hit in remote cache, demand data read, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_REMOTE_RD](#) that accesses a previously modified cache location.

DSNP_HITM_REMOTE_RW, Snoop hit in remote cache, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_REMOTE_RW](#) that accesses a previously modified cache location.

DSNP_HITM_REMOTE_ST_RETIRED, Store instruction architecturally executed, snoop hit in remote cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HITM_REMOTE_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

DSNP_HITM_REMOTE_WR, Snoop hit in remote cache, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_REMOTE_WR](#) that accesses a previously modified cache location.

DSNP_HITM_RW, Snoop hit, demand data access, modified

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that accesses a previously modified cache location.

DSNP_HITM_ST_RETIRED, Store instruction architecturally executed, snoop hit

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HITM_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

DSNP_HITM_WR, Snoop hit, demand data write, modified

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_WR](#) that accesses a previously modified cache location.

DSNP_HIT_FAR, Snoop hit in far local cache, data access

The counter counts each snoop counted by [DSNP_HIT](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_FAR_HWPRF, Snoop hit in far local cache, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by [DSNP_HIT_HWPRF](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_FAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in far cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [DSNP_HIT_LDST_RETIRED](#) that hit outside the local PE cluster on the same device.

DSNP_HIT_FAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in far cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HIT_FAR_LDST_RETIRED](#) that is a Memory-reading instruction.

DSNP_HIT_FAR_PRF, Snoop hit in far local cache, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by [DSNP_HIT_PRF](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_FAR_PRFM, Snoop hit in far local cache, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by [DSNP_HIT_PRFM](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_FAR_RD, Snoop hit in far local cache, demand data read

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_RD](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_FAR_RW, Snoop hit in far local cache, demand data access

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_FAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in far cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HIT_FAR_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

DSNP_HIT_FAR_WR, Snoop hit in far local cache, demand data write

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_WR](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_HWPRF, Snoop hit, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by [DSNP_HIT_PRF](#) that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that hit in a data cache outside the cache hierarchy of this PE.

DSNP_HIT_LD_RETIRED, Load instruction architecturally executed, snoop hit

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

DSNP_HIT_N1, Snoop hit in cache at distance 1, data access

The counter counts each snoop counted by [DSNP_HIT](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N1_HWPRF, Snoop hit in cache at distance 1, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by [DSNP_HIT_HWPRF](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N1_PRF, Snoop hit in cache at distance 1, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by [DSNP_HIT_PRF](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N1_PRFM, Snoop hit in cache at distance 1, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by [DSNP_HIT_PRFM](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N1_RD, Snoop hit in cache at distance 1, demand data read

The counter counts each snoop generated by the PE in response to a demand 'Memory-read operation' counted by [DSNP_HIT_RD](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N1_RW, Snoop hit in cache at distance 1, demand data access

The counter counts each snoop generated by the PE in response to a demand 'Memory-read operation' or demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N1_WR, Snoop hit in cache at distance 1, demand data write

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_WR](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N2, Snoop hit in cache at distance 2, data access

The counter counts each snoop counted by [DSNP_HIT](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N2_HWPRF, Snoop hit in cache at distance 2, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by [DSNP_HIT_HWPRF](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N2_PRF, Snoop hit in cache at distance 2, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by [DSNP_HIT_PRF](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N2_PRFM, Snoop hit in cache at distance 2, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by [DSNP_HIT_PRFM](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N2_RD, Snoop hit in cache at distance 2, demand data read

The counter counts each snoop generated by the PE in response to a demand 'Memory-read operation' counted by [DSNP_HIT_RD](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N2_RW, Snoop hit in cache at distance 2, demand data access

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N2_WR, Snoop hit in cache at distance 2, demand data write

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_WR](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N3, Snoop hit in cache at distance 3, data access

The counter counts each snoop counted by [DSNP_HIT](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N3_HWPRF, Snoop hit in cache at distance 3, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by [DSNP_HIT_HWPRF](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N3_PRF, Snoop hit in cache at distance 3, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by [DSNP_HIT_PRF](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N3_PRFM, Snoop hit in cache at distance 3, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by [DSNP_HIT_PRFM](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N3_RD, Snoop hit in cache at distance 3, demand data read

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_RD](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N3_RW, Snoop hit in cache at distance 3, demand data access

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N3_WR, Snoop hit in cache at distance 3, demand data write

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_WR](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N4, Snoop hit in cache at distance 4, data access

The counter counts each snoop counted by [DSNP_HIT](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N4_HWPRF, Snoop hit in cache at distance 4, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by [DSNP_HIT_HWPRF](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N4_PRF, Snoop hit in cache at distance 4, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by [DSNP_HIT_PRF](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N4_PRFM, Snoop hit in cache at distance 4, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by [DSNP_HIT_PRFM](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N4_RD, Snoop hit in cache at distance 4, demand data read

The counter counts each snoop generated by the PE in response to a demand 'Memory-read operation' counted by [DSNP_HIT_RD](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N4_RW, Snoop hit in cache at distance 4, demand data access

The counter counts each snoop generated by the PE in response to a demand 'Memory-read operation' or demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_N4_WR, Snoop hit in cache at distance 4, demand data write

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_WR](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

DSNP_HIT_NEAR, Snoop hit in near local cache, data access

The counter counts each snoop counted by [DSNP_HIT](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_NEAR_HWPRF, Snoop hit in near local cache, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by [DSNP_HIT_HWPRF](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_NEAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in near cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [DSNP_HIT_LDST_RETIRED](#) that hit outside of the cache hierarchy of this PE in the local PE cluster.

DSNP_HIT_NEAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in near cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HIT_NEAR_LDST_RETIRED](#) that is a Memory-reading instruction.

DSNP_HIT_NEAR_PRF, Snoop hit in near local cache, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by [DSNP_HIT_PRF](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_NEAR_PRFM, Snoop hit in near local cache, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by [DSNP_HIT_PRFM](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_NEAR_RD, Snoop hit in near local cache, demand data read

The counter counts each snoop generated by the PE in response to a demand 'Memory-read operation' counted by [DSNP_HIT_RD](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_NEAR_RW, Snoop hit in near local cache, demand data access

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_NEAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in near cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HIT_NEAR_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

DSNP_HIT_NEAR_WR, Snoop hit in near local cache, demand data write

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_WR](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_PRF, Snoop hit, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_PRFM, Snoop hit, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by [DSNP_HIT_PRF](#) that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_RD, Snoop hit, demand data read

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_RW](#) that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_REMOTE, Snoop hit in remote cache, data access

The counter counts each snoop counted by [DSNP_HIT](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_REMOTE_HWPRF, Snoop hit in remote cache, hardware data prefetch

The counter counts each snoop generated by a data hardware prefetch counted by [DSNP_HIT_HWPRF](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_REMOTE_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in remote cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [DSNP_HIT_LDST_RETIRED](#) that hit on a remote device.

DSNP_HIT_REMOTE_LD_RETIRED, Load instruction architecturally executed, snoop hit in remote cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HIT_REMOTE_LDST_RETIRED](#) that is a Memory-reading instruction.

DSNP_HIT_REMOTE_PRF, Snoop hit in remote cache, data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by [DSNP_HIT_PRF](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_REMOTE_PRFM, Snoop hit in remote cache, software data prefetch

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by [DSNP_HIT_PRFM](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_REMOTE_RD, Snoop hit in remote cache, demand data read

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ counted by [DSNP_HIT_RD](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_REMOTE_RW, Snoop hit in remote cache, demand data access

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_REMOTE_ST_RETIRED, Store instruction architecturally executed, snoop hit in remote cache

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HIT_REMOTE_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

DSNP_HIT_REMOTE_WR, Snoop hit in remote cache, demand data write

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_WR](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_RW, Snoop hit, demand data access

The counter counts each snoop generated by the PE in response to a demand ‘Memory-read operation’ or demand [Memory-write operation](#) that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

DSNP_HIT_ST_RETIRED, Store instruction architecturally executed, snoop hit

The counter counts each [architecturally executed](#) instruction counted by [DSNP_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

DSNP_HIT_WR, Snoop hit, demand data write

The counter counts each snoop generated by the PE in response to a demand [Memory-write operation](#) counted by [DSNP_HIT_RW](#) that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

DTLB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, no translation table walk

The counter counts each Memory-reading instruction or [Memory-writing instruction architecturally executed](#) counted by [LDST_ANY_RETIRED](#) that did not cause a refill of a data TLB involving at least one translation table walk access.

DTLB_HIT_LD_RETIRED, Load instruction architecturally executed, no translation table walk

The counter counts each Memory-reading instruction [architecturally executed](#) counted by [DTLB_HIT_LDST_RETIRED](#) that did not cause a refill of a data TLB involving at least one translation table walk access.

DTLB_HIT_ST_RETIRED, Store instruction architecturally executed, no translation table walk

The counter counts each [Memory-writing instruction architecturally executed](#) counted by [DTLB_HIT_LDST_RETIRED](#) that did not cause a refill of a data TLB involving at least one translation table walk access.

DTLB_WALK_HWPRF, Data TLB hardware prefetch, with at least one translation table walk

The counter counts each hardware prefetch access counted by [L1D_TLB_HWPRF](#) that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

DTLB_WALK_LDST_RETIRED, Load or store instruction architecturally executed, at least one translation table walk

The counter counts each Memory-reading instruction or [Memory-writing instruction architecturally executed](#) counted by [LDST_ANY_RETIRED](#) that caused a refill of a data TLB involving at least one translation table walk access.

The counter does not count the Memory-reading instruction or [Memory-writing instruction architecturally executed](#) if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the Memory-reading instruction or [Memory-writing instruction architecturally executed](#) if the access generates a Translation fault for any other reason.

DTLB_WALK_LD_RETIRED, Load instruction architecturally executed, at least one translation table walk

The counter counts each Memory-reading instruction [architecturally executed](#) counted by [DTLB_WALK_LDST_RETIRED](#) that caused a refill of a data TLB involving at least one translation table walk access.

The counter does not count the Memory-reading instruction [architecturally executed](#) if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the Memory-reading instruction [architecturally executed](#) if the access generates a Translation fault for any other reason.

DTLB_WALK_PERCYC, Data translation table walks in progress

The counter increments by the number of data translation table walks counted by [DTLB_WALK](#) in progress on each [Processor cycle](#).

The ratio $\text{DTLB_WALK_PERCYC} \div \text{DTLB_WALK}$ is the mean duration of data translation table walks in [Processor cycles](#).

If the TLB is shared and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPERS<n>_EL0.MT.

DTLB_WALK_PRF, Data TLB prefetch, with at least one translation table walk

The counter counts each software prefetch or hardware prefetch access counted by [L1D_TLB_PRF](#) that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the software prefetch or hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch or hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPERS<n>_EL0.MT.

DTLB_WALK_PRFM, Data TLB software prefetch, with at least one translation table walk

The counter counts each software prefetch access counted by [L1D_TLB_PRFM](#) that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the software prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPERS<n>_EL0.MT

for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

DTLB_WALK_RW, Data TLB demand access, with at least one translation table walk

The counter counts each demand access counted by [L1D_TLB_RW](#) that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the demand access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the demand access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

DTLB_WALK_ST_RETIRED, Store instruction architecturally executed, at least one translation table walk

The counter counts each [Memory-writing instruction architecturally executed](#) counted by [DTLB_WALK_LDST_RETIRED](#) that caused a refill of a data TLB involving at least one translation table walk access.

The counter does not count the [Memory-writing instruction architecturally executed](#) if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the [Memory-writing instruction architecturally executed](#) if the access generates a Translation fault for any other reason.

FPASE_LDST_REG_SPEC, Operation speculatively executed, SIMD&FP register load or store

The counter counts each [speculatively executed](#) operation that reads from or writes to memory due to any of the following A64 instructions:

- Scalar: LDP (SIMD&FP), LDR (SIMD&FP), STP (SIMD&FP), or STR (SIMD&FP).
- Advanced SIMD: LD1, LD1R, LDNP, LDP, LDR, LDUR, ST1, STNP, STP, STR, or STUR.

It is IMPLEMENTATION DEFINED which load and store operations are counted in AArch32 state.

FPASE_LD_REG_SPEC, Operation speculatively executed, SIMD&FP register load

The counter counts each [speculatively executed](#) operation that reads from memory counted by [FPASE_LDST_REG_SPEC](#) due to any of the following A64 instructions:

- Scalar: LDP (SIMD&FP) or LDR (SIMD&FP).
- Advanced SIMD: LD1, LD1R, LDNP, LDP, LDR, or LDUR.

It is IMPLEMENTATION DEFINED which load operations are counted in AArch32 state.

FPASE_ST_REG_SPEC, Operation speculatively executed, SIMD&FP register store

The counter counts each [speculatively executed](#) operation that writes to memory counted by [FPASE_LDST_REG_SPEC](#) due to any of the following A64 instructions:

- Scalar: STP (SIMD&FP) or STR (SIMD&FP).
- Advanced SIMD: ST1, STNP, STP, STR, or STUR.

It is IMPLEMENTATION DEFINED which store operations are counted in AArch32 state.

FP_ADDSUB_SPEC, Floating-point operation speculatively executed, add or subtract

The counter counts each [speculatively executed](#) floating-point add or subtract operation counted by [FP_SPEC](#) due to any of the following A64 instructions:

- Scalar: FADD or FSUB.
- Advanced SIMD: FABD, FADD, or FSUB.

It is IMPLEMENTATION DEFINED which floating-point add or subtract operations are counted in AArch32 state.

FP_BF16_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is BFloat16 floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the smallest type was BFloat16 floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by [FP_SCALE_OPS_SPEC](#).

FP_BF16_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is BFloat16 floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the largest type was BFloat16 floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by [FP_SCALE_OPS_SPEC](#).

FP_BF16_MIN_SPEC, Floating-point operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each floating-point operation counted by [FP_SPEC](#) due to an instruction where the smallest type was BFloat16 floating-point.

FP_CVT_SPEC, Floating-point operation speculatively executed, convert

The counter counts each [speculatively executed](#) floating-point convert operation due to any of:

- An A64 scalar floating-point conversion instruction.
- An Advanced SIMD floating-point conversion instruction.
- An SVE floating-point conversion instruction.

This includes both conversions between floating-point types, and conversions between integer and floating-point types.

It is IMPLEMENTATION DEFINED which floating-point convert operations are counted in AArch32 state.

FP_DIV_SPEC, Floating-point operation speculatively executed, divide

The counter counts each [speculatively executed](#) floating-point divide operation counted by [FP_SPEC](#) due to any of the following A64 instructions:

- Scalar: FDIV.
- Advanced SIMD: FDIV.

It is IMPLEMENTATION DEFINED which floating-point divide operations are counted in AArch32 state.

FP_DP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is double-precision floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the largest type was double-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

FP_DP_SPEC, Floating-point operation speculatively executed, double-precision, data-processing

The counter counts each [speculatively executed](#) floating-point data-processing operation counted by [FP_SPEC](#) where the largest type is double-precision.

FP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the type was floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

FP_FMA_SPEC, Floating-point operation speculatively executed, FMA

The counter counts each [speculatively executed](#) floating-point fused multiply-add or multiply-subtract operation counted by [FP_SPEC](#) due to any of the following A64 instructions:

- Scalar: FMADD, FMSUB, FNMADD, or FNMSUB.
- Advanced SIMD: FMLA or FMLS.

It is IMPLEMENTATION DEFINED which floating-point fused multiply-add or multiply-subtract operations are counted in AArch32 state.

FP_FP8_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is 8-bit floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the smallest type was 8-bit floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

FP_FP8_MIN_SPEC, Floating-point operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each floating-point operation counted by [FP_SPEC](#) due to an instruction where the smallest type was 8-bit floating-point.

FP_HP_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is half-precision floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the smallest type was half-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by [FP_SCALE_OPS_SPEC](#).

FP_HP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is half-precision floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the largest type was half-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by [FP_SCALE_OPS_SPEC](#).

FP_HP_MIN_SPEC, Floating-point operation speculatively executed, smallest type is half-precision floating-point

The counter counts each floating-point operation counted by [FP_SPEC](#) due to an instruction where the smallest type was half-precision floating-point.

FP_HP_SPEC, Floating-point operation speculatively executed, half-precision

The counter counts each [speculatively executed](#) floating-point operation counted by [FP_SPEC](#) where the largest type is half-precision.

FP_MUL_SPEC, Floating-point operation speculatively executed, multiply

The counter counts each [speculatively executed](#) floating-point multiply operation counted by [FP_SPEC](#) due to any of the following A64 instructions:

- Scalar: FMUL or FMULX.
- Advanced SIMD: FMUL or FMULX.

It is IMPLEMENTATION DEFINED which floating-point multiply operations are counted in AArch32 state.

FP_RECPE_SPEC, Floating-point operation speculatively executed, reciprocal estimate

The counter counts each [speculatively executed](#) floating-point reciprocal estimate operation counted by [FP_SPEC](#) due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRTE.

It is IMPLEMENTATION DEFINED which floating-point reciprocal estimate operations are counted in AArch32 state.

FP_SPEC, Floating-point operation speculatively executed, including SIMD

The counter counts each [speculatively executed](#) floating-point data-processing operation due to any of:

- An A64 scalar instruction.
- An A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts any operation due to the following instructions:

- Scalar: `FABS`, or `FNEG` (scalar).
- Advanced SIMD: `FABS`, `FNEG` (vector), `FRECPE`, or `FRSQRT`.
- Data movement instructions which move data without carrying out any operations on the data. This includes identity data-processing operations.
- Conditional select instructions.
- Floating-point conversion instructions.

Any operation counted by [INT_SPEC](#) is not counted.

It is IMPLEMENTATION DEFINED which floating-point data-processing operations are counted in AArch32 state.

Arm recommends that if [FP_CVT_SPEC](#) is implemented, [FP_SPEC](#) counts operations due to convert instructions.

Note

This event differs from the [VFP_SPEC](#) event which does not count SIMD operations.

FP_SP_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is single-precision floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the smallest type was single-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by [FP_SCALE_OPS_SPEC](#).

FP_SP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is single-precision floating-point

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the largest type was single-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by [FP_SCALE_OPS_SPEC](#).

FP_SP_MIN_SPEC, Floating-point operation speculatively executed, smallest type is single-precision floating-point

The counter counts each floating-point operation counted by [FP_SPEC](#) due to an instruction where the smallest type was single-precision floating-point.

FP_SP_SPEC, Floating-point operation speculatively executed, single-precision

The counter counts each [speculatively executed](#) floating-point operation counted by [FP_SPEC](#) where the largest type is single-precision.

FP_SQRT_SPEC, Floating-point operation speculatively executed, square root

The counter counts each [speculatively executed](#) floating-point square-root operation counted by [FP_SPEC](#) due to any of the following A64 instructions:

- Scalar: FSQRT.
- Advanced SIMD: FSQRT.

It is IMPLEMENTATION DEFINED which floating-point square-root operations are counted in AArch32 state.

GCSTLB_WALK, GCS TLB access with at least one translation table walk

The counter counts each access counted by [LIGCS_TLB](#) that causes a refill of a TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

Accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access is due to a TLB maintenance instruction.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPERS<n>_EL0.MT.

GCSTLB_WALK_PERCYC, Translation table walks in progress

The counter increments by the number of translation table walks counted by [GCSTLB_WALK](#) in progress on each [Processor cycle](#).

The ratio $\text{GCSTLB_WALK_PERCYC} \div \text{GCSTLB_WALK}$ is the mean duration of translation table walks in [Processor cycles](#).

If the TLB is shared and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPERS<n>_EL0.MT.

GCSTLB_WALK_RW, GCS TLB demand access with at least one translation table walk

The counter counts each demand access counted by both [GCSTLB_WALK](#) and [LIGCS_TLB_RW](#).

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

GCSTLB_WALK_STEP, GCS TLB translation table walk, step

The counter counts each translation table walk access due to a refill of a TLB.

The translation table walk must be due to either GCS data read operation or GCS data write operation.

The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

The counter does not count the event if any of the following are true:

- The access causing the refill generates a Translation fault because the applicable `TCR_ELx.EPDy` bit is 1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable `TCR_ELx.EOPDy` bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the event if the access causing the refill generates a Translation fault for any other reason.

Implementation of this optional event requires that `FEAT_EOPD` is implemented.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

INST_FETCH, Instruction memory access

The counter counts each [Instruction memory access](#).

The counter increments whether the access results in an access to a Level 1 instruction cache, a Level 2 instruction, data or unified cache, or none of these.

The counter does not increment as a result of:

- Data memory accesses.
- Translation table walks.
- Refilling of any cache.
- Accesses that result from cache maintenance instructions.

INST_FETCH_PERCYC, Instruction fetches in progress

The counter increments by the number of instruction fetches counted by [INST_FETCH](#) in progress on each [Processor cycle](#).

The ratio $\text{INST_FETCH_PERCYC} \div \text{INST_FETCH}$ is the mean duration of instruction fetches in [Processor cycles](#).

INST_FETCH_PRFM, Instruction memory access, software prefetch

The counter counts each fetch counted by [INST_FETCH](#) that is due to a prefetch instruction.

INST_FETCH_RD, Instruction memory access, demand fetch

The counter counts each fetch counted by `INST_FETCH` that is a demand [Instruction memory access](#).

This includes any fetch made for a speculative instruction.

INT_DIV64_SPEC, Integer operation speculatively executed, 64-bit divide

The counter counts each [speculatively executed](#) integer divide operation counted by `INT_SPEC` due to any of the following A64 instructions:

- Scalar: SDIV or UDIV.

The counter only counts operations with 64-bit operands or vector elements.

INT_DIV_SPEC, Integer operation speculatively executed, divide

The counter counts each [speculatively executed](#) integer divide operation counted by `INT_SPEC` due to any of the following A64 instructions:

- Scalar: SDIV or UDIV.

It is IMPLEMENTATION DEFINED which integer divide operations are counted in AArch32 state.

INT_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, integer

The counter increments by v for each [speculatively executed](#) non-scalable element arithmetic operation, due to an instruction where the type was integer.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by `FP_SCALE_OPS_SPEC`.

INT_MUL64_SPEC, Integer operation speculatively executed, 64×64 multiply

The counter counts each [speculatively executed](#) 64×64 integer multiply operation counted by `INT_SPEC` due to any of the following A64 instructions:

- Scalar: MADD, MSUB, MUL, SMULH, or UMULH.

The counter only counts operations that perform a $64\text{-bit} \times 64\text{-bit}$ integer multiply operations.

INT_MULH64_SPEC, Integer operation speculatively executed, 64×64 multiply returning high part

The counter counts each [speculatively executed](#) widening 64×64 integer multiply operation counted by `INT_SPEC` due to any of the following A64 instructions:

- Scalar: SMULH or UMULH.

These instructions perform $64\text{-bit} \times 64\text{-bit}$ integer multiply operations.

INT_MUL_SPEC, Integer operation speculatively executed, multiply

The counter counts each [speculatively executed](#) integer multiply or multiply-accumulate operation counted by [INT_SPEC](#) due to any of the following A64 instructions:

- Scalar: MADD, MSUB, MUL, SMADDL, SMULH, UMADDL, or UMULH.
- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.

It is IMPLEMENTATION DEFINED which integer multiply or multiply-accumulate operations are counted in AArch32 state.

INT_SPEC, Integer operation speculatively executed

The counter counts each [speculatively executed](#) integer arithmetic operation due to any of:

- An A64 scalar data-processing instruction.
- An Advanced SIMD instruction.
- An SVE data-processing instruction.

It is IMPLEMENTATION DEFINED whether the counter counts any operation due to the following instructions:

- Scalar: FABS, FNEG (scalar).
- Advanced SIMD: FABS, FNEG (vector), FRECPE, FRSQRTE.
- Data movement instructions which move data without carrying out any operations on the data. This includes identity data-processing operations.
- Conditional select instructions.
- Floating-point conversion instructions.

Any operation counted by [FP_SPEC](#) is not counted.

It is IMPLEMENTATION DEFINED which integer arithmetic operations are counted in AArch32 state.

ISNP_HIT, Snoop hit, instruction

The counter counts each snoop that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_FAR, Snoop hit in far local cache, instruction access

The counter counts each snoop counted by [ISNP_HIT](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_FAR_HWPRF, Snoop hit in far local cache, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by [ISNP_HIT_HWPRF](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_FAR_PRF, Snoop hit in far local cache, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by [ISNP_HIT_PRF](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_FAR_PRFM, Snoop hit in far local cache, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by [ISNP_HIT_PRFM](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_FAR_RD, Snoop hit in far local cache, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand [Instruction memory access](#) counted by [ISNP_HIT_RD](#) that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_FAR_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in far cache

The counter counts each [instruction architecturally executed](#) counted by [ISNP_HIT_RETIRED](#) that hit outside the local PE cluster on the same device when fetched.

ISNP_HIT_HWPRF, Snoop hit, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by [ISNP_HIT_PRF](#) that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_N1, Snoop hit in cache at distance 1, instruction access

The counter counts each snoop counted by [ISNP_HIT](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N1_HWPRF, Snoop hit in cache at distance 1, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by [ISNP_HIT_HWPRF](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N1_PRF, Snoop hit in cache at distance 1, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by [ISNP_HIT_PRF](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N1_PRFM, Snoop hit in cache at distance 1, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by [ISNP_HIT_PRFM](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N1_RD, Snoop hit in cache at distance 1, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand [Instruction memory access](#) counted by [ISNP_HIT_RD](#) that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N2, Snoop hit in cache at distance 2, instruction access

The counter counts each snoop counted by [ISNP_HIT](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N2_HWPRF, Snoop hit in cache at distance 2, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by [ISNP_HIT_HWPRF](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N2_PRF, Snoop hit in cache at distance 2, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by [ISNP_HIT_PRF](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N2_PRFM, Snoop hit in cache at distance 2, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by [ISNP_HIT_PRFM](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N2_RD, Snoop hit in cache at distance 2, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand [Instruction memory access](#) counted by [ISNP_HIT_RD](#) that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N3, Snoop hit in cache at distance 3, instruction access

The counter counts each snoop counted by [ISNP_HIT](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N3_HWPRF, Snoop hit in cache at distance 3, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by [ISNP_HIT_HWPRF](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N3_PRF, Snoop hit in cache at distance 3, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by [ISNP_HIT_PRF](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N3_PRFM, Snoop hit in cache at distance 3, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by [ISNP_HIT_PRFM](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N3_RD, Snoop hit in cache at distance 3, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand [Instruction memory access](#) counted by [ISNP_HIT_RD](#) that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N4, Snoop hit in cache at distance 4, instruction access

The counter counts each snoop counted by [ISNP_HIT](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N4_HWPRF, Snoop hit in cache at distance 4, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by [ISNP_HIT_HWPRF](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N4_PRF, Snoop hit in cache at distance 4, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by [ISNP_HIT_PRF](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N4_PRFM, Snoop hit in cache at distance 4, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by [ISNP_HIT_PRFM](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_N4_RD, Snoop hit in cache at distance 4, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand [Instruction memory access](#) counted by [ISNP_HIT_RD](#) that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

ISNP_HIT_NEAR, Snoop hit in near local cache, instruction access

The counter counts each snoop counted by [ISNP_HIT](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_NEAR_HWPRF, Snoop hit in near local cache, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by [ISNP_HIT_HWPRF](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_NEAR_PRF, Snoop hit in near local cache, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by [ISNP_HIT_PRF](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_NEAR_PRFM, Snoop hit in near local cache, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by [ISNP_HIT_PRFM](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_NEAR_RD, Snoop hit in near local cache, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand [Instruction memory access](#) counted by [ISNP_HIT_RD](#) that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_NEAR_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in near cache

The counter counts each [instruction architecturally executed](#) counted by [ISNP_HIT_RETIRED](#) that hit outside of the cache hierarchy of this PE in the local PE cluster when fetched.

ISNP_HIT_PRF, Snoop hit, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_PRFM, Snoop hit, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by [ISNP_HIT_PRF](#) that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_RD, Snoop hit, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand [Instruction memory access](#) that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_REMOTE, Snoop hit in remote cache, instruction access

The counter counts each snoop counted by [ISNP_HIT](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_REMOTE_HWPRF, Snoop hit in remote cache, hardware instruction prefetch

The counter counts each snoop generated by an instruction hardware prefetch counted by [ISNP_HIT_HWPRF](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_REMOTE_PRF, Snoop hit in remote cache, instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by [ISNP_HIT_PRF](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_REMOTE_PRFM, Snoop hit in remote cache, software instruction prefetch

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by [ISNP_HIT_PRFM](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_REMOTE_RD, Snoop hit in remote cache, demand instruction fetch

The counter counts each snoop generated by the PE in response to a demand [Instruction memory access](#) counted by [ISNP_HIT_RD](#) that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

ISNP_HIT_REMOTE_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in remote cache

The counter counts each [instruction architecturally executed](#) counted by [ISNP_HIT_RETIRED](#) that hit on a remote device when fetched.

ISNP_HIT_RETIRED, Instruction architecturally executed, instruction fetch snoop hit

The counter counts each [instruction architecturally executed](#) counted by [INST_RETIRED](#) that hit in an instruction cache outside the cache hierarchy of this PE when fetched.

ITLB_HIT_RETIRED, Instruction architecturally executed, no translation table walk

The counter counts each instruction which was [architecturally executed](#) counted by [INST_RETIRED](#) that did not cause a refill of an instruction TLB involving at least one translation table walk access.

The counter does not count the instruction which was [architecturally executed](#) if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the instruction which was [architecturally executed](#) if the access generates a Translation fault for any other reason.

ITLB_WALK_HWPRF, Instruction TLB hardware prefetch, with at least one translation table walk

The counter counts each hardware prefetch access counted by [L1I_TLB_HWPRF](#) that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

ITLB_WALK_PERCYC, Instruction translation table walks in progress

The counter increments by the number of instruction translation table walks counted by [ITLB_WALK](#) in progress on each [Processor cycle](#).

The ratio $\text{ITLB_WALK_PERCYC} \div \text{ITLB_WALK}$ is the mean duration of instruction translation table walks in [Processor cycles](#).

If the TLB is shared and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPEPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPEPER<n>_EL0.MT.

ITLB_WALK_PRF, Instruction TLB prefetch, with at least one translation table walk

The counter counts each software prefetch or hardware prefetch access counted by [L1I_TLB_PRF](#) that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the software prefetch or hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch or hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPERS<n>_EL0.MT.

ITLB_WALK_PRFM, Instruction TLB software prefetch, with at least one translation table walk

The counter counts each software prefetch access counted by [L1I_TLB_PRFM](#) that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the software prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPERS<n>_EL0.MT.

ITLB_WALK_RD, Instruction TLB demand access, with at least one translation table walk

The counter counts each demand access counted by [L1I_TLB_RD](#) that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the demand access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the demand access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPERS<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPERS<n>_EL0.MT.

ITLB_WALK_RETIRED, Instruction architecturally executed, at least one translation table walk

The counter counts each instruction which was [architecturally executed](#) counted by [INST_RETIRED](#) that caused a refill of an instruction TLB involving at least one translation table walk access.

The counter does not count the instruction which was [architecturally executed](#) if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the instruction which was [architecturally executed](#) if the access generates a Translation fault for any other reason.

L1D_CACHE_HIT, Level 1 data cache hit

The counter counts each access counted by [L1D_CACHE](#) that hits in the Level 1 data or unified cache.

L1D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 1 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [L1D_CACHE_HIT_LDST_RETIRED](#) that hit modified data in Level 1 data or unified cache.

L1D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 1 data cache

The counter counts each [architecturally executed](#) instruction counted by [L1D_CACHE_HITM_LDST_RETIRED](#) that is a Memory-reading instruction.

L1D_CACHE_HITM_RD, Level 1 data cache demand access hit modified, read

The counter counts each cache access counted by [L1D_CACHE_HIT_RD](#) that accesses a previously modified cache location.

L1D_CACHE_HITM_RW, Level 1 data cache demand access hit modified

The counter counts each cache access counted by [L1D_CACHE_HIT_RW](#) that accesses a previously modified cache location.

L1D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 1 data cache

The counter counts each [architecturally executed](#) instruction counted by [L1D_CACHE_HITM_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L1D_CACHE_HITM_WR, Level 1 data cache demand access hit modified, write

The counter counts each cache access counted by [L1D_CACHE_HIT_WR](#) that accesses a previously modified cache location.

L1D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 1 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that hit in Level 1 data or unified cache.

L1D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 1 data cache

The counter counts each [architecturally executed](#) instruction counted by [L1D_CACHE_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

L1D_CACHE_HIT_PRFM, Level 1 data cache software prefetch hit

The counter counts each software prefetch counted by [L1D_CACHE_PRFM](#) that hits in the Level 1 data or unified cache.

L1D_CACHE_HIT_RD, Level 1 data cache demand access hit, read

The counter counts each demand read counted by [L1D_CACHE_RD](#) that hits in the Level 1 data or unified cache.

L1D_CACHE_HIT_RD_FHWPRF, Level 1 data cache demand access first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by [L1D_CACHE_HIT_RW_FHWPRF](#) that is due to a demand ‘Memory-read operation’.

L1D_CACHE_HIT_RD_FPRF, Level 1 data cache demand access first hit, read, fetched by prefetch

The counter counts each first hit counted by [L1D_CACHE_HIT_RW_FPRF](#) that is due to a demand ‘Memory-read operation’.

L1D_CACHE_HIT_RD_FPRFM, Level 1 data cache demand access first hit, read, fetched by software prefetch

The counter counts each first hit counted by [L1D_CACHE_HIT_RW_FPRFM](#) that is due to a demand ‘Memory-read operation’.

L1D_CACHE_HIT_RW, Level 1 data cache demand access hit

The counter counts each demand access counted by [L1D_CACHE_RW](#) that hits in the Level 1 data or unified cache.

L1D_CACHE_HIT_RW_FHWPRF, Level 1 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by [L1D_CACHE_HIT_RW_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the [L1D_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1D_CACHE_HIT_RW_FPRF, Level 1 data cache demand access first hit, fetched by prefetch

The counter counts each demand access first hit counted by [L1D_CACHE_HIT_RW](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the [L1D_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1D_CACHE_HIT_RW_FPRFM, Level 1 data cache demand access first hit, fetched by software prefetch

The counter counts each demand access first hit counted by [L1D_CACHE_HIT_RW_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the [L1D_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 1 data cache

The counter counts each [architecturally executed](#) instruction counted by [L1D_CACHE_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L1D_CACHE_HIT_WR, Level 1 data cache demand access hit, write

The counter counts each demand write counted by [L1D_CACHE_WR](#) that hits in the Level 1 data or unified cache.

L1D_CACHE_HIT_WR_FHWPRF, Level 1 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by [L1D_CACHE_HIT_RW_FHWPRF](#) that is due to a demand [Memory-write operation](#).

L1D_CACHE_HIT_WR_FPRF, Level 1 data cache demand access first hit, write, fetched by prefetch

The counter counts each first hit counted by [L1D_CACHE_HIT_RW_FPRF](#) that is due to a demand [Memory-write operation](#).

L1D_CACHE_HIT_WR_FPRFM, Level 1 data cache demand access first hit, write, fetched by software prefetch

The counter counts each first hit counted by [L1D_CACHE_HIT_RW_FPRFM](#) that is due to a demand [Memory-write operation](#).

L1D_CACHE_HWPRF, Level 1 data cache hardware prefetch

The counter counts each access counted by [L1D_CACHE_PRF](#) that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 1 data or unified cache.

L1D_CACHE_MISS, Level 1 data cache demand access miss

The counter counts each demand access counted by [L1D_CACHE_RW](#) that misses in the Level 1 data or unified cache, causing an access to outside of the Level 1 caches of this PE.

L1D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 1 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that missed in Level 1 data or unified cache.

L1D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 1 data cache

The counter counts each [architecturally executed](#) instruction counted by [L1D_CACHE_MISS_LDST_RETIRED](#) that is a Memory-reading instruction.

L1D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 1 data cache

The counter counts each [architecturally executed](#) instruction counted by [L1D_CACHE_MISS_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L1D_CACHE_PRF, Level 1 data cache, prefetch hit

The counter counts each access counted by [L1D_CACHE](#) that is due to a prefetch instruction, or hardware prefetch.

L1D_CACHE_PRFM, Level 1 data cache software prefetch

The counter counts each access counted by [L1D_CACHE_PRF](#) that is due to a prefetch instruction.

Arm recommends that this event is implemented if event [L1D_CACHE_RW](#) is implemented.

L1D_CACHE_REFILL_HWPRF, Level 1 data cache refill, hardware prefetch

The counter counts each hardware prefetch counted by [L1D_CACHE_HWPRF](#) that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

L1D_CACHE_REFILL_PERCYC, Level 1 data or unified cache refills in progress

The counter increments by the number of Level 1 data or unified cache refills counted by [L1D_CACHE_REFILL](#) in progress on each [Processor cycle](#).

The ratio $\text{L1D_CACHE_REFILL_PERCYC} \div \text{L1D_CACHE_REFILL}$ is the mean duration of Level 1 data or unified cache refills in [Processor cycles](#).

L1D_CACHE_REFILL_PRF, Level 1 data cache refill, prefetch hit

The counter counts each access counted by [L1D_CACHE_PRF](#) that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

L1D_CACHE_REFILL_PRFM, Level 1 data cache refill, software prefetch

The counter counts each access counted by [L1D_CACHE_PRFM](#) that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

L1D_CACHE_RW, Level 1 data cache demand access

The counter counts each access counted by [L1D_CACHE](#) that is due to a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

This includes accesses made by [speculatively executed](#) instructions.

Arm recommends that this event is implemented if any of the following are true:

- Event [L1D_CACHE_PRFM](#) is implemented.
- Event [L1D_CACHE_HWPRF](#) is implemented.

L1D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 1 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [L1D_CACHE_HIT_LDST_RETIRED](#) that hit a recently fetched line in Level 1 data or unified cache.

L1D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 1 data cache

The counter counts each [architecturally executed](#) instruction counted by [L1D_LFB_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

L1D_LFB_HIT_RD, Level 1 data cache demand access line-fill buffer hit, read

The counter counts each demand access counted by [L1D_CACHE_HIT_RD](#) that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L1D_LFB_HIT_RD_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by [L1D_LFB_HIT_RW_FHWPRF](#) that is due to a demand ‘Memory-read operation’.

L1D_LFB_HIT_RD_FPRF, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by [L1D_LFB_HIT_RW_FPRF](#) that is due to a demand ‘Memory-read operation’.

L1D_LFB_HIT_RD_FPRFM, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by [L1D_LFB_HIT_RW_FPRFM](#) that is due to a demand ‘Memory-read operation’.

L1D_LFB_HIT_RW, Level 1 data cache demand access line-fill buffer hit

The counter counts each demand access counted by [L1D_CACHE_HIT_RW](#) that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L1D_LFB_HIT_RW_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by [L1D_LFB_HIT_RW_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L1D_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1D_LFB_HIT_RW_FPRF, Level 1 data cache demand access line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand access line-fill buffer first hit counted by [L1D_LFB_HIT_RW](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L1D_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1D_LFB_HIT_RW_FPRFM, Level 1 data cache demand access line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand access line-fill buffer first hit counted by [L1D_LFB_HIT_RW_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L1D_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 1 data cache

The counter counts each [architecturally executed](#) instruction counted by [L1D_LFB_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L1D_LFB_HIT_WR, Level 1 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by [L1D_CACHE_HIT_WR](#) that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L1D_LFB_HIT_WR_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by [L1D_LFB_HIT_RW_FHWPRF](#) that is due to a demand [Memory-write operation](#).

L1D_LFB_HIT_WR_FPRF, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by [L1D_LFB_HIT_RW_FPRF](#) that is due to a demand [Memory-write operation](#).

L1D_LFB_HIT_WR_FPRFM, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by [L1D_LFB_HIT_RW_FPRFM](#) that is due to a demand [Memory-write operation](#).

L1D_TLB_HWPRF, Level 1 data TLB access, hardware prefetch

The counter counts each access counted by [L1D_TLB_PRF](#) that is due to a hardware prefetch.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

L1D_TLB_PRF, Level 1 data TLB access, prefetch

The counter counts each access counted by [L1D_TLB](#) that is due to a prefetch instruction or hardware prefetch.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

L1D_TLB_PRFM, Level 1 data TLB access, software prefetch

The counter counts each access counted by [L1D_TLB_PRF](#) that is due to a prefetch instruction.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

Arm recommends that this event is implemented if event [L1D_TLB_RW](#) is implemented.

L1D_TLB_RW, Level 1 data TLB access, demand access

The counter counts each access counted by [L1D_TLB](#) that is due to a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

Arm recommends that this event is implemented if event [L1D_TLB_PRFM](#) is implemented.

L1GCS_CACHE, Level 1 GCS cache access

The counter counts each GCS data read operation or GCS data write operation that causes a cache access to at least a Level 1 cache.

Each access to a cache line is counted including the multiple accesses caused by single instructions such as `GCSPUSHX` or `GCSPOPX`. Each access to other Level 1 memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

Accesses to a Level 1 cache due to a GCS hardware prefetcher as well are counted.

When there are separate GCS caching structures implemented:

- It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.
- Accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

L1GCS_CACHE_HIT_RW, Level 1 GCS cache demand access hit

The counter counts each access counted by [L1GCS_CACHE_RW](#) that hits in a Level 1 cache.

L1GCS_CACHE_HIT_RW_FHWPRF, Level 1 GCS cache demand access first hit, fetched by GCS hardware prefetcher

The counter counts each demand access first hit counted by [L1GCS_CACHE_HIT_RW](#) where the cache line was fetched by a GCS hardware prefetcher.

That is, the [L1GCS_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1GCS_CACHE_HWPRF, Level 1 GCS cache hardware prefetch

The counter counts each access counted by [L1GCS_CACHE](#) that is due to a GCS hardware prefetcher.

L1GCS_CACHE_INVALID, Level 1 GCS cache invalidate

The counter counts each invalidation of a cache line in a Level 1 GCS cache.

For example:

- Invalidation of a cache line because of a GCSB effect.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 1 cache invalidates a line in the Level 1 cache.

Note

Software that uses this event must know whether the Level 1 cache is shared with other PEs. This event does not follow the general rule of Level 1 cache events of only counting Attributable events.

L1GCS_CACHE_MISS, Level 1 GCS cache demand access miss

The counter counts each access counted by [L1GCS_CACHE_RW](#) that misses in the Level 1 caches causing an access to outside of the Level 1 caches of this PE.

L1GCS_CACHE_MISS_RD, Level 1 GCS cache demand access read miss

The counter counts each access counted by [L1GCS_CACHE_MISS](#) that is also a Memory-read operation.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

Arm recommends that this event is implemented if event [L1GCS_CACHE](#) is implemented.

L1GCS_CACHE_REFILL, Level 1 GCS cache refill

The counter counts each access counted by [L1GCS_CACHE](#) that causes a refill of a Level 1 cache from outside of the Level 1 cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

L1GCS_CACHE_REFILL_HWPRF, Level 1 GCS cache refill, hardware prefetch

The counter counts each hardware prefetch counted by [L1GCS_CACHE_HWPRF](#) that causes a refill of a Level 1 cache from outside of the Level 1 cache.

L1GCS_CACHE_REFILL_PERCYC, Level 1 GCS cache refills in progress

The counter increments by the number of Level 1 GCS cache refills counted by [L1GCS_CACHE_REFILL](#) in progress on each [Processor cycle](#).

The ratio $\text{L1GCS_CACHE_REFILL_PERCYC} \div \text{L1GCS_CACHE_REFILL}$ is the mean duration of Level 1 GCS cache refills in [Processor cycles](#).

L1GCS_CACHE_RW, Level 1 GCS cache demand access

The counter counts each access counted by [L1GCS_CACHE](#) that is due to a demand Memory-read operation or demand Memory-write operation.

This includes accesses made by Speculatively executed instructions.

Arm recommends that this event is implemented if event [L1GCS_CACHE_HWPRF](#) is implemented.

L1GCS_LFB_HIT_RW, Level 1 GCS cache demand access line-fill buffer hit

The counter counts each demand access counted by [L1GCS_CACHE_HIT_RW](#) that hits a recently fetched line in a Level 1 cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L1GCS_LFB_HIT_RW_FHWPRF, Level 1 GCS cache demand access line-fill buffer first hit, recently fetched by GCS hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by [L1GCS_LFB_HIT_RW](#) where the cache line was fetched by a GCS hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L1GCS_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

L1GCS_TLB, Level 1 GCS TLB access

The counter counts each GCS data read operation or GCS data write operation that causes a TLB access to at least a Level 1 TLB.

Each access to a TLB entry is counted including multiple accesses caused by single instructions such as [GCSPUSHX](#) or [GCSPOPX](#).

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of [PMEVTYPER<n>_EL0.MT](#) for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of [PMEVTYPER<n>_EL0.MT](#) for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of [PMEVTYPER<n>_EL0.MT](#).

L1GCS_TLB_HWPRF, Level 1 GCS demand TLB access, GCS hardware prefetch

The counter counts each access counted by [L1GCS_TLB](#) that is due to a GCS hardware prefetch.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

L1GCS_TLB_RW, Level 1 GCS demand TLB access

The counter counts each access counted by [L1GCS_TLB](#) that is due to a demand Memory-read operation or demand Memory-write operation.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

L1I_CACHE_HIT, Level 1 instruction cache hit

The counter counts each access counted by [L1I_CACHE](#) that hits in the Level 1 instruction or unified cache.

L1I_CACHE_HIT_PRFM, Level 1 instruction cache software prefetch hit

The counter counts each software prefetch counted by [L1I_CACHE_PRFM](#) that hits in the Level 1 instruction or unified cache.

L1I_CACHE_HIT_RD, Level 1 instruction cache demand fetch hit

The counter counts each demand fetch counted by [L1I_CACHE_RD](#) that hits in the Level 1 instruction or unified cache.

L1I_CACHE_HIT_RD_FHWPRF, Level 1 instruction cache demand fetch first hit, fetched by hardware prefetcher

The counter counts each demand fetch first hit counted by [L1I_CACHE_HIT_RD_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the [L1I_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1I_CACHE_HIT_RD_FPRF, Level 1 instruction cache demand fetch first hit, fetched by prefetch

The counter counts each demand fetch first hit counted by [L1I_CACHE_HIT_RD](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the [L1I_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1I_CACHE_HIT_RD_FPRFM, Level 1 instruction cache demand fetch first hit, fetched by software prefetch

The counter counts each demand fetch first hit counted by [L1I_CACHE_HIT_RD_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the [L1I_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1I_CACHE_HIT_RETIRED, Instruction architecturally executed, hit in Level 1 instruction cache

The counter counts each [instruction architecturally executed](#) counted by [INST_RETIRED](#) that hit in Level 1 instruction or unified cache when fetched.

L1I_CACHE_HWPRF, Level 1 instruction cache hardware prefetch

The counter counts each access counted by [L1I_CACHE_PRF](#) that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 1 instruction or unified cache.

The PE might prefetch instructions as part of instruction pipelining, and might do so for [speculatively executed](#) instruction paths. PE instruction prefetching is not counted as hardware prefetching. Unlike PE instruction prefetching, a hardware prefetch only allocates instructions into the cache.

L1I_CACHE_LMISS, Level 1 instruction cache long-latency miss

If the [L1I_CACHE_RD](#) event is implemented, the counter counts each access to the Level 1 instruction or unified cache counted by [L1I_CACHE_RD](#) that incurs additional latency because it returns instructions from outside of the Level 1 instruction or unified cache of this PE.

If the [L1I_CACHE_RD](#) event is not implemented, the counter counts each access to the Level 1 instruction or unified cache counted by [L1I_CACHE](#) that incurs additional latency because it returns instructions from outside of the Level 1 instruction or unified cache of this PE.

The event indicates to software that the access missed in the Level 1 instruction or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 instruction or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 1 caches that the access looks up in and results in instructions being returned from a Level 2 cache or elsewhere beyond the Level 1 instruction or unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[38]` reads as 1 if this event is implemented and 0 otherwise.

L1I_CACHE_MISS_RETIRED, Instruction architecturally executed, miss in Level 1 instruction cache

The counter counts each [instruction architecturally executed](#) counted by [INST_RETIRED](#) that missed in Level 1 instruction or unified cache when fetched.

L1I_CACHE_PRF, Level 1 instruction cache, prefetch hit

The counter counts each access counted by [L1I_CACHE](#) that is due to a prefetch instruction, or hardware prefetch.

L1I_CACHE_PRFM, Level 1 instruction cache software prefetch

The counter counts each access counted by [L1I_CACHE_PRF](#) that is due to a prefetch instruction.

Arm recommends that this event is implemented if event [L1I_CACHE_RD](#) is implemented.

L1I_CACHE_RD, Level 1 instruction cache demand fetch

The counter counts each access counted by [L1I_CACHE](#) that is due to a demand [Instruction memory access](#).

This includes instruction prefetches made by the PE for [speculatively executed](#) instructions.

Arm recommends that this event is implemented if any of the following are true:

- Event [L1I_CACHE_PRFM](#) is implemented.
- Event [L1I_CACHE_HWPRF](#) is implemented.

L1I_CACHE_REFILL_HWPRF, Level 1 instruction cache refill, hardware prefetch

The counter counts each hardware prefetch counted by [L1I_CACHE_HWPRF](#) that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

L1I_CACHE_REFILL_PERCYC, Level 1 instruction or unified cache refills in progress

The counter increments by the number of Level 1 instruction or unified cache refills counted by [L1I_CACHE_REFILL](#) in progress on each [Processor cycle](#).

The ratio $\text{L1I_CACHE_REFILL_PERCYC} \div \text{L1I_CACHE_REFILL}$ is the mean duration of Level 1 instruction or unified cache refills in [Processor cycles](#).

L1I_CACHE_REFILL_PRF, Level 1 instruction cache refill, prefetch hit

The counter counts each access counted by [L1I_CACHE_PRF](#) that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

L1I_CACHE_REFILL_PRFM, Level 1 instruction cache refill, software prefetch

The counter counts each access counted by [L1I_CACHE_PRFM](#) that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

L1I_LFB_HIT_RD, Level 1 instruction cache demand fetch line-fill buffer hit

The counter counts each demand access counted by [L1I_CACHE_HIT_RD](#) that hits a recently fetched line in the Level 1 instruction or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L1I_LFB_HIT_RD_FHWPFRF, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand fetch line-fill buffer first hit counted by [L1I_LFB_HIT_RD_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L1I_CACHE_REFILL_HWPFRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1I_LFB_HIT_RD_FPRF, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand fetch line-fill buffer first hit counted by [L1I_LFB_HIT_RD](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L1I_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1I_LFB_HIT_RD_FPRFM, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand fetch line-fill buffer first hit counted by [L1I_LFB_HIT_RD_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L1I_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L1I_LFB_HIT_RETIRED, Instruction architecturally executed, line-fill buffer hit in Level 1 instruction cache

The counter counts each [instruction architecturally executed](#) counted by [L1I_CACHE_HIT_RETIRED](#) that hit a recently fetched line in Level 1 instruction or unified cache when fetched.

L1I_TLB_HWPFRF, Level 1 instruction TLB access, hardware prefetch

The counter counts each access counted by [L1I_TLB_PRF](#) that is due to a hardware prefetch.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

L1I_TLB_PRF, Level 1 instruction TLB access, prefetch

The counter counts each access counted by [L1I_TLB](#) that is due to a prefetch instruction or hardware prefetch.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other

cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

L1I_TLB_PRFM, Level 1 instruction TLB access, software prefetch

The counter counts each access counted by `L1I_TLB_PRF` that is due to a prefetch instruction.

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

Arm recommends that this event is implemented if event `L1I_TLB_RD` is implemented.

L1I_TLB_RD, Level 1 instruction TLB access, demand access

The counter counts each access counted by `L1I_TLB` that is due to a demand [Instruction memory access](#).

If the TLB is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

Arm recommends that this event is implemented if event `L1I_TLB_PRFM` is implemented.

L2D_CACHE_HIT, Level 2 data cache hit

The counter counts each access counted by `L2D_CACHE` that hits in the Level 2 data or unified cache.

L2D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 2 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by `L2D_CACHE_HIT_LDST_RETIRED` that hit modified data in Level 2 data or unified cache.

L2D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 2 data cache

The counter counts each [architecturally executed](#) instruction counted by `L2D_CACHE_HITM_LDST_RETIRED` that is a Memory-reading instruction.

L2D_CACHE_HITM_RD, Level 2 data cache demand access hit modified, read

The counter counts each cache access counted by `L2D_CACHE_HIT_RD` that accesses a previously modified cache location.

L2D_CACHE_HITM_RW, Level 2 data cache demand access hit modified

The counter counts each cache access counted by `L2D_CACHE_HIT_RW` that accesses a previously modified cache location.

L2D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 2 data cache

The counter counts each [architecturally executed](#) instruction counted by `L2D_CACHE_HITM_LDST_RETIRED` that is a [Memory-writing instruction](#).

L2D_CACHE_HITM_WR, Level 2 data cache demand access hit modified, write

The counter counts each cache access counted by [L2D_CACHE_HIT_WR](#) that accesses a previously modified cache location.

L2D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 2 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that hit in Level 2 data or unified cache.

L2D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 2 data cache

The counter counts each [architecturally executed](#) instruction counted by [L2D_CACHE_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

L2D_CACHE_HIT_PRFM, Level 2 data cache software prefetch hit

The counter counts each software prefetch counted by [L2D_CACHE_PRFM](#) that hits in the Level 2 data or unified cache.

L2D_CACHE_HIT_RD, Level 2 data cache demand access hit, read

The counter counts each demand read counted by [L2D_CACHE_RD](#) that hits in the Level 2 data or unified cache.

L2D_CACHE_HIT_RD_FHWPRF, Level 2 data cache demand access first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by [L2D_CACHE_HIT_RW_FHWPRF](#) that is due to a demand ‘Memory-read operation’.

L2D_CACHE_HIT_RD_FPRF, Level 2 data cache demand access first hit, read, fetched by prefetch

The counter counts each first hit counted by [L2D_CACHE_HIT_RW_FPRF](#) that is due to a demand ‘Memory-read operation’.

L2D_CACHE_HIT_RD_FPRFM, Level 2 data cache demand access first hit, read, fetched by software prefetch

The counter counts each first hit counted by [L2D_CACHE_HIT_RW_FPRFM](#) that is due to a demand ‘Memory-read operation’.

L2D_CACHE_HIT_RW, Level 2 data cache demand access hit

The counter counts each demand access counted by [L2D_CACHE_RW](#) that hits in the Level 2 data or unified cache.

L2D_CACHE_HIT_RW_FHWPRF, Level 2 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by [L2D_CACHE_HIT_RW_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the [L2D_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2D_CACHE_HIT_RW_FPRF, Level 2 data cache demand access first hit, fetched by prefetch

The counter counts each demand access first hit counted by [L2D_CACHE_HIT_RW](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the [L2D_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2D_CACHE_HIT_RW_FPRFM, Level 2 data cache demand access first hit, fetched by software prefetch

The counter counts each demand access first hit counted by [L2D_CACHE_HIT_RW_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the [L2D_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 2 data cache

The counter counts each [architecturally executed](#) instruction counted by [L2D_CACHE_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L2D_CACHE_HIT_WR, Level 2 data cache demand access hit, write

The counter counts each demand write counted by [L2D_CACHE_WR](#) that hits in the Level 2 data or unified cache.

L2D_CACHE_HIT_WR_FHWPRF, Level 2 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by [L2D_CACHE_HIT_RW_FHWPRF](#) that is due to a demand [Memory-write operation](#).

L2D_CACHE_HIT_WR_FPRF, Level 2 data cache demand access first hit, write, fetched by prefetch

The counter counts each first hit counted by [L2D_CACHE_HIT_RW_FPRF](#) that is due to a demand [Memory-write operation](#).

L2D_CACHE_HIT_WR_FPRFM, Level 2 data cache demand access first hit, write, fetched by software prefetch

The counter counts each first hit counted by [L2D_CACHE_HIT_RW_FPRFM](#) that is due to a demand [Memory-write operation](#).

L2D_CACHE_HWPRF, Level 2 data cache hardware prefetch

The counter counts each access counted by [L2D_CACHE_PRF](#) that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 2 data or unified cache.

L2D_CACHE_LMISS_RD, Level 2 data cache long-latency read miss

The counter counts each ‘Memory-read operation’ to the Level 2 data or unified cache counted by [L2D_CACHE_RD](#) that incurs additional latency because it returns data from outside of the Level 1 and Level 2 data or unified caches of this PE.

The event indicates to software that the access missed in the Level 2 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 2 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 2 caches that the access looks up in and results in an access being made to a Level 3 cache or elsewhere beyond the Level 2 data or unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[41]` reads as 1 if this event is implemented and 0 otherwise.

L2D_CACHE_MISS, Level 2 data cache demand access miss

The counter counts each demand access counted by [L2D_CACHE_RW](#) that misses in the Level 1 and Level 2 data or unified caches, causing an access to outside of the Level 1 and Level 2 caches of this PE.

L2D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 2 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that missed in Level 2 data or unified cache.

L2D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 2 data cache

The counter counts each [architecturally executed](#) instruction counted by [L2D_CACHE_MISS_LDST_RETIRED](#) that is a Memory-reading instruction.

L2D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 2 data cache

The counter counts each [architecturally executed](#) instruction counted by [L2D_CACHE_MISS_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L2D_CACHE_PRF, Level 2 data cache, prefetch hit

The counter counts each access counted by [L2D_CACHE](#) that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 2 data or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

L2D_CACHE_PRFM, Level 2 data cache software prefetch

The counter counts each access counted by [L2D_CACHE_PRF](#) that is due to a prefetch instruction.

This includes accesses to the Level 2 data or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event [L2D_CACHE_RW](#) is implemented.

L2D_CACHE_REFILL_HWPRF, Level 2 data cache refill, hardware prefetch

The counter counts each hardware prefetch counted by [L2D_CACHE_HWPRF](#) that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

L2D_CACHE_REFILL_PERCYC, Level 2 data or unified cache refills in progress

The counter increments by the number of Level 2 data or unified cache refills counted by [L2D_CACHE_REFILL](#) in progress on each [Processor cycle](#).

The ratio $\text{L2D_CACHE_REFILL_PERCYC} \div \text{L2D_CACHE_REFILL}$ is the mean duration of Level 2 data or unified cache refills in [Processor cycles](#).

L2D_CACHE_REFILL_PRF, Level 2 data cache refill, prefetch hit

The counter counts each access counted by [L2D_CACHE_PRF](#) that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

L2D_CACHE_REFILL_PRFM, Level 2 data cache refill, software prefetch

The counter counts each access counted by [L2D_CACHE_PRFM](#) that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

L2D_CACHE_RW, Level 2 data cache demand access

The counter counts each access counted by [L2D_CACHE](#) that is due to a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

This includes:

- Accesses made by [speculatively executed](#) instructions.
- Accesses to the Level 2 data or unified cache due to a refill of another cache caused by a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

Arm recommends that this event is implemented if any of the following are true:

- Event [L2D_CACHE_PRFM](#) is implemented.
- Event [L2D_CACHE_HWPFR](#) is implemented.

L2D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 2 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [L2D_CACHE_HIT_LDST_RETIRED](#) that hit a recently fetched line in Level 2 data or unified cache.

L2D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 2 data cache

The counter counts each [architecturally executed](#) instruction counted by [L2D_LFB_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

L2D_LFB_HIT_RD, Level 2 data cache demand access line-fill buffer hit, read

The counter counts each demand access counted by [L2D_CACHE_HIT_RD](#) that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L2D_LFB_HIT_RD_FHWPFR, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by [L2D_LFB_HIT_RW_FHWPFR](#) that is due to a demand ‘Memory-read operation’.

L2D_LFB_HIT_RD_FPRF, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by [L2D_LFB_HIT_RW_FPRF](#) that is due to a demand ‘Memory-read operation’.

L2D_LFB_HIT_RD_FPRFM, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by [L2D_LFB_HIT_RW_FPRFM](#) that is due to a demand ‘Memory-read operation’.

L2D_LFB_HIT_RW, Level 2 data cache demand access line-fill buffer hit

The counter counts each demand access counted by [L2D_CACHE_HIT_RW](#) that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L2D_LFB_HIT_RW_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by [L2D_LFB_HIT_RW_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L2D_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2D_LFB_HIT_RW_FPRF, Level 2 data cache demand access line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand access line-fill buffer first hit counted by [L2D_LFB_HIT_RW](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L2D_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2D_LFB_HIT_RW_FPRFM, Level 2 data cache demand access line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand access line-fill buffer first hit counted by [L2D_LFB_HIT_RW_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L2D_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 2 data cache

The counter counts each [architecturally executed](#) instruction counted by [L2D_LFB_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L2D_LFB_HIT_WR, Level 2 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by [L2D_CACHE_HIT_WR](#) that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L2D_LFB_HIT_WR_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by [L2D_LFB_HIT_RW_FHWPRF](#) that is due to a demand [Memory-write operation](#).

L2D_LFB_HIT_WR_FPRF, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by [L2D_LFB_HIT_RW_FPRF](#) that is due to a demand [Memory-write operation](#).

L2D_LFB_HIT_WR_FPRFM, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by [L2D_LFB_HIT_RW_FPRFM](#) that is due to a demand [Memory-write operation](#).

L2I_CACHE_HIT, Level 2 instruction cache hit

The counter counts each access counted by [L2I_CACHE](#) that hits in the Level 2 instruction or unified cache.

L2I_CACHE_HIT_PRFM, Level 2 instruction cache software prefetch hit

The counter counts each software prefetch counted by [L2I_CACHE_PRFM](#) that hits in the Level 2 instruction or unified cache.

L2I_CACHE_HIT_RD, Level 2 instruction cache demand fetch hit

The counter counts each demand fetch counted by [L2I_CACHE_RD](#) that hits in the Level 2 instruction or unified cache.

L2I_CACHE_HIT_RD_FHWPRF, Level 2 instruction cache demand fetch first hit, fetched by hardware prefetcher

The counter counts each demand fetch first hit counted by [L2I_CACHE_HIT_RD_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the [L2I_CACHE_REFILL_HWPFRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2I_CACHE_HIT_RD_FPRF, Level 2 instruction cache demand fetch first hit, fetched by prefetch

The counter counts each demand fetch first hit counted by [L2I_CACHE_HIT_RD](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the [L2I_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2I_CACHE_HIT_RD_FPRFM, Level 2 instruction cache demand fetch first hit, fetched by software prefetch

The counter counts each demand fetch first hit counted by [L2I_CACHE_HIT_RD_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the [L2I_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2I_CACHE_HIT_RETIRED, Instruction architecturally executed, hit in Level 2 instruction cache

The counter counts each [instruction architecturally executed](#) counted by [INST_RETIRED](#) that hit in Level 2 instruction or unified cache when fetched.

L2I_CACHE_HWPRF, Level 2 instruction cache hardware prefetch

The counter counts each access counted by [L2I_CACHE_PRF](#) that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 2 instruction or unified cache.

L2I_CACHE_LMISS, Level 2 instruction cache long-latency miss

If the [L2I_CACHE_RD](#) event is implemented, the counter counts each access to the Level 2 instruction or unified cache counted by [L2I_CACHE_RD](#) that incurs additional latency because it returns instructions from outside of the Level 1 and Level 2 instruction or unified caches of this PE.

If the [L2I_CACHE_RD](#) event is not implemented, the counter counts each access to the Level 2 instruction or unified cache counted by [L2I_CACHE](#) that incurs additional latency because it returns instructions from outside of the Level 1 and Level 2 instruction or unified caches of this PE.

The event indicates to software that the access missed in the Level 2 instruction or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 2 instruction or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 2 caches that the access looks up in and results in instructions being returned from a Level 3 cache or elsewhere beyond the Level 2 instruction or unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[42]` reads as 1 if this event is implemented and 0 otherwise.

L2I_CACHE_MISS_RETIRED, Instruction architecturally executed, miss in Level 2 instruction cache

The counter counts each [instruction architecturally executed](#) counted by [INST_RETIRED](#) that missed in Level 2 instruction or unified cache when fetched.

L2I_CACHE_PRF, Level 2 instruction cache, prefetch hit

The counter counts each access counted by [L2I_CACHE](#) that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

L2I_CACHE_PRFM, Level 2 instruction cache software prefetch

The counter counts each access counted by [L2I_CACHE_PRF](#) that is due to a prefetch instruction.

This includes accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event [L2I_CACHE_RD](#) is implemented.

L2I_CACHE_RD, Level 2 instruction cache demand fetch

The counter counts each access counted by [L2I_CACHE](#) that is due to a demand [Instruction memory access](#).

This includes:

- Instruction prefetches made by the PE for [speculatively executed](#) instructions.
- Accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a demand [Instruction memory access](#).

Arm recommends that this event is implemented if any of the following are true:

- Event [L2I_CACHE_PRFM](#) is implemented.
- Event [L2I_CACHE_HWPRF](#) is implemented.

L2I_CACHE_REFILL_HWPRF, Level 2 instruction cache refill, hardware prefetch

The counter counts each hardware prefetch counted by [L2I_CACHE_HWPRF](#) that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

L2I_CACHE_REFILL_PERCYC, Level 2 instruction or unified cache refills in progress

The counter increments by the number of Level 2 instruction or unified cache refills counted by [L2I_CACHE_REFILL](#) in progress on each [Processor cycle](#).

The ratio $\text{L2I_CACHE_REFILL_PERCYC} \div \text{L2I_CACHE_REFILL}$ is the mean duration of Level 2 instruction or unified cache refills in [Processor cycles](#).

L2I_CACHE_REFILL_PRF, Level 2 instruction cache refill, prefetch hit

The counter counts each access counted by [L2I_CACHE_PRF](#) that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

L2I_CACHE_REFILL_PRFM, Level 2 instruction cache refill, software prefetch

The counter counts each access counted by [L2I_CACHE_PRFM](#) that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

L2I_LFB_HIT_RD, Level 2 instruction cache demand fetch line-fill buffer hit

The counter counts each demand access counted by [L2I_CACHE_HIT_RD](#) that hits a recently fetched line in the Level 2 instruction or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L2I_LFB_HIT_RD_FHWPRF, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand fetch line-fill buffer first hit counted by [L2I_LFB_HIT_RD_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L2I_CACHE_REFILL_HWPFRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2I_LFB_HIT_RD_FPRF, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand fetch line-fill buffer first hit counted by [L2I_LFB_HIT_RD](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L2I_CACHE_REFILL_PRFRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2I_LFB_HIT_RD_FPRFM, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand fetch line-fill buffer first hit counted by [L2I_LFB_HIT_RD_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L2I_CACHE_REFILL_PRFRM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L2I_LFB_HIT_RETIRED, Instruction architecturally executed, line-fill buffer hit in Level 2 instruction cache

The counter counts each [instruction architecturally executed](#) counted by [L2I_CACHE_HIT_RETIRED](#) that hit a recently fetched line in Level 2 instruction or unified cache when fetched.

L3D_CACHE_HIT, Level 3 data cache hit

The counter counts each access counted by [L3D_CACHE](#) that hits in the Level 3 data or unified cache.

L3D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 3 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [L3D_CACHE_HIT_LDST_RETIRED](#) that hit modified data in Level 3 data or unified cache.

L3D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 3 data cache

The counter counts each [architecturally executed](#) instruction counted by [L3D_CACHE_HITM_LDST_RETIRED](#) that is a Memory-reading instruction.

L3D_CACHE_HITM_RD, Level 3 data cache demand access hit modified, read

The counter counts each cache access counted by [L3D_CACHE_HIT_RD](#) that accesses a previously modified cache location.

L3D_CACHE_HITM_RW, Level 3 data cache demand access hit modified

The counter counts each cache access counted by [L3D_CACHE_HIT_RW](#) that accesses a previously modified cache location.

L3D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 3 data cache

The counter counts each [architecturally executed](#) instruction counted by [L3D_CACHE_HITM_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L3D_CACHE_HITM_WR, Level 3 data cache demand access hit modified, write

The counter counts each cache access counted by [L3D_CACHE_HIT_WR](#) that accesses a previously modified cache location.

L3D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 3 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that hit in Level 3 data or unified cache.

L3D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 3 data cache

The counter counts each [architecturally executed](#) instruction counted by [L3D_CACHE_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

L3D_CACHE_HIT_PRFM, Level 3 data cache software prefetch hit

The counter counts each software prefetch counted by [L3D_CACHE_PRFM](#) that hits in the Level 3 data or unified cache.

L3D_CACHE_HIT_RD, Level 3 data cache demand access hit, read

The counter counts each demand read counted by [L3D_CACHE_RD](#) that hits in the Level 3 data or unified cache.

L3D_CACHE_HIT_RD_FHWPRF, Level 3 data cache demand access first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by [L3D_CACHE_HIT_RW_FHWPRF](#) that is due to a demand ‘Memory-read operation’.

L3D_CACHE_HIT_RD_FPRF, Level 3 data cache demand access first hit, read, fetched by prefetch

The counter counts each first hit counted by [L3D_CACHE_HIT_RW_FPRF](#) that is due to a demand ‘Memory-read operation’.

L3D_CACHE_HIT_RD_FPRFM, Level 3 data cache demand access first hit, read, fetched by software prefetch

The counter counts each first hit counted by [L3D_CACHE_HIT_RW_FPRFM](#) that is due to a demand ‘Memory-read operation’.

L3D_CACHE_HIT_RW, Level 3 data cache demand access hit

The counter counts each demand access counted by [L3D_CACHE_RW](#) that hits in the Level 3 data or unified cache.

L3D_CACHE_HIT_RW_FHWPRF, Level 3 data cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by [L3D_CACHE_HIT_RW_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the [L3D_CACHE_REFILL_HWPFRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L3D_CACHE_HIT_RW_FPRF, Level 3 data cache demand access first hit, fetched by prefetch

The counter counts each demand access first hit counted by [L3D_CACHE_HIT_RW](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the [L3D_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L3D_CACHE_HIT_RW_FPRFM, Level 3 data cache demand access first hit, fetched by software prefetch

The counter counts each demand access first hit counted by [L3D_CACHE_HIT_RW_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the [L3D_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L3D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 3 data cache

The counter counts each [architecturally executed](#) instruction counted by [L3D_CACHE_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L3D_CACHE_HIT_WR, Level 3 data cache demand access hit, write

The counter counts each demand write counted by [L3D_CACHE_WR](#) that hits in the Level 3 data or unified cache.

L3D_CACHE_HIT_WR_FHWPRF, Level 3 data cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by [L3D_CACHE_HIT_RW_FHWPRF](#) that is due to a demand [Memory-write operation](#).

L3D_CACHE_HIT_WR_FPRF, Level 3 data cache demand access first hit, write, fetched by prefetch

The counter counts each first hit counted by [L3D_CACHE_HIT_RW_FPRF](#) that is due to a demand [Memory-write operation](#).

L3D_CACHE_HIT_WR_FPRFM, Level 3 data cache demand access first hit, write, fetched by software prefetch

The counter counts each first hit counted by [L3D_CACHE_HIT_RW_FPRFM](#) that is due to a demand [Memory-write operation](#).

L3D_CACHE_HWPRF, Level 3 data cache hardware prefetch

The counter counts each access counted by [L3D_CACHE_PRF](#) that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 3 data or unified cache.

L3D_CACHE_LMISS_RD, Level 3 data cache long-latency read miss

The counter counts each ‘Memory-read operation’ to the Level 3 data or unified cache counted by [L3D_CACHE_RD](#) that incurs additional latency because it returns data from outside of the Level 1 to Level 3 data or unified caches of this PE.

The event indicates to software that the access missed in the Level 3 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 3 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 3 caches that the access looks up in and results in an access being made to a lower level cache or elsewhere beyond the Level 3 data or unified cache. This might be counted as a Level 2 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

`PMCEID0_EL0[43]` reads as 1 if this event is implemented and 0 otherwise.

L3D_CACHE_MISS, Level 3 data cache demand access miss

The counter counts each demand access counted by [L3D_CACHE_RW](#) that misses in the Level 1 to Level 3 data or unified caches, causing an access to outside of the Level 1 to Level 3 caches of this PE.

L3D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 3 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that missed in Level 3 data or unified cache.

L3D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 3 data cache

The counter counts each [architecturally executed](#) instruction counted by [L3D_CACHE_MISS_LDST_RETIRED](#) that is a Memory-reading instruction.

L3D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 3 data cache

The counter counts each [architecturally executed](#) instruction counted by [L3D_CACHE_MISS_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L3D_CACHE_PRF, Level 3 data cache, prefetch hit

The counter counts each access counted by [L3D_CACHE](#) that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 3 data or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

L3D_CACHE_PRFM, Level 3 data cache software prefetch

The counter counts each access counted by [L3D_CACHE_PRF](#) that is due to a prefetch instruction.

This includes accesses to the Level 3 data or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event [L3D_CACHE_RW](#) is implemented.

L3D_CACHE_REFILL_HWPRF, Level 3 data cache refill, hardware prefetch

The counter counts each hardware prefetch counted by [L3D_CACHE_HWPRF](#) that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

L3D_CACHE_REFILL_PERCYC, Level 3 data or unified cache refills in progress

The counter increments by the number of Level 3 data or unified cache refills counted by [L3D_CACHE_REFILL](#) in progress on each [Processor cycle](#).

The ratio $\text{L3D_CACHE_REFILL_PERCYC} \div \text{L3D_CACHE_REFILL}$ is the mean duration of Level 3 data or unified cache refills in [Processor cycles](#).

L3D_CACHE_REFILL_PRF, Level 3 data cache refill, prefetch hit

The counter counts each access counted by [L3D_CACHE_PRF](#) that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

L3D_CACHE_REFILL_PRFM, Level 3 data cache refill, software prefetch

The counter counts each access counted by [L3D_CACHE_PRFM](#) that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

L3D_CACHE_RW, Level 3 data cache demand access

The counter counts each access counted by [L3D_CACHE](#) that is due to a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

This includes:

- Accesses made by [speculatively executed](#) instructions.
- Accesses to the Level 3 data or unified cache due to a refill of another cache caused by a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

Arm recommends that this event is implemented if any of the following are true:

- Event [L3D_CACHE_PRFM](#) is implemented.
- Event [L3D_CACHE_HWPRF](#) is implemented.

L3D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 3 data cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [L3D_CACHE_HIT_LDST_RETIRED](#) that hit a recently fetched line in Level 3 data or unified cache.

L3D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 3 data cache

The counter counts each [architecturally executed](#) instruction counted by [L3D_LFB_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

L3D_LFB_HIT_RD, Level 3 data cache demand access line-fill buffer hit, read

The counter counts each demand access counted by [L3D_CACHE_HIT_RD](#) that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L3D_LFB_HIT_RD_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by [L3D_LFB_HIT_RW_FHWPRF](#) that is due to a demand 'Memory-read operation'.

L3D_LFB_HIT_RD_FPRF, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by [L3D_LFB_HIT_RW_FPRF](#) that is due to a demand 'Memory-read operation'.

L3D_LFB_HIT_RD_FPRFM, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by [L3D_LFB_HIT_RW_FPRFM](#) that is due to a demand 'Memory-read operation'.

L3D_LFB_HIT_RW, Level 3 data cache demand access line-fill buffer hit

The counter counts each demand access counted by [L3D_CACHE_HIT_RW](#) that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L3D_LFB_HIT_RW_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by [L3D_LFB_HIT_RW_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L3D_CACHE_REFILL_HWPFRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L3D_LFB_HIT_RW_FPRF, Level 3 data cache demand access line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand access line-fill buffer first hit counted by [L3D_LFB_HIT_RW](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L3D_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L3D_LFB_HIT_RW_FPRFM, Level 3 data cache demand access line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand access line-fill buffer first hit counted by [L3D_LFB_HIT_RW_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [L3D_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

L3D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 3 data cache

The counter counts each [architecturally executed](#) instruction counted by [L3D_LFB_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

L3D_LFB_HIT_WR, Level 3 data cache demand access line-fill buffer hit, write

The counter counts each demand access counted by [L3D_CACHE_HIT_WR](#) that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

L3D_LFB_HIT_WR_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by [L3D_LFB_HIT_RW_FHWPRF](#) that is due to a demand [Memory-write operation](#).

L3D_LFB_HIT_WR_FPRF, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by [L3D_LFB_HIT_RW_FPRF](#) that is due to a demand [Memory-write operation](#).

L3D_LFB_HIT_WR_FPRFM, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by [L3D_LFB_HIT_RW_FPRFM](#) that is due to a demand [Memory-write operation](#).

LDST_ALIGN_LAT, Access with additional latency from alignment

The counter counts each access counted by [MEM_ACCESS](#) that incurred additional latency due to the alignment of the address and size of data being accessed.

PMCEID1_EL0[32] reads as 1 if this event is implemented and 0 otherwise.

LDST_FIXED_BYTES_SPEC, Non-scalable load and store bytes speculatively executed

The counter counts each byte speculatively read or written due to any of:

- Any load, store, or atomic operation, other than loads and stores of the SVE P and Z registers.
- Any SVE non-vector load or store operation.
- An SVE load and replicate instruction, including: LD1R, LD1RB, LD1RD, LD1RH, LD1RQ, LD1RQB, LD1RQD, LD1RQH, LD1RQW, LD1RSB, LD1RSH, or LD1RSW.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred multiplied by the number of transfers made per register. For example, the counter counts bytes as follows:

- Non-SVE load and store of a single register instructions increment the counter by $(MSIZE \div 8)$. This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- Non-SVE load and store of a pair of registers instructions increment the counter by $2 \times (MSIZE \div 8)$. This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- AArch32 load and store multiple registers instructions increment the counter by the number of registers transferred multiplied by $(MSIZE \div 8)$.
- Atomic store instructions increment the counter by $(MSIZE \div 8)$. These are instructions that atomically update a value in memory without returning a value to a register.
- Atomic load, compare and swap of a single register, and swap instructions increment the counter by $2 \times (MSIZE \div 8)$. Atomic load instructions are instructions that atomically update a value in memory, returning a value to a register.
- Compare and swap of a pair of registers increment the counter by $4 \times (MSIZE \div 8)$.
- Advanced SIMD LD1R instructions increment the counter by $(MSIZE \div 8)$.
- Advanced SIMD LD[1-4] and ST[1-4] instructions increment the counter by the number of registers transferred multiplied by the number of bytes being transferred per register.
- DC ZVA instructions increment by the counter by $2^{(DCZID_EL0.BS)}$.

LDST_FIXED_OPS_SPEC, Non-scalable load or store element operations speculatively executed

The counter counts each [speculatively executed](#) ‘Memory-read operation’ or [Memory-write operation](#) due to any of:

- Any load, store, or atomic operation, other than loads and stores of the SVE P and Z registers.
- Any SVE non-vector load or store operation.
- An SVE load and replicate instruction, including: LD1R, LD1RB, LD1RD, LD1RH, LD1RQ, LD1RQB, LD1RQD, LD1RQH, LD1RQW, LD1RSB, LD1RSH, or LD1RSW.

For each instruction, the counter is incremented by the number of operations specified by the instruction. For example, the counter counts operations as follows:

- Non-SVE load and store of a single register instructions increment the counter by 1. This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- Non-SVE load and store of a pair of registers instructions increment the counter by 2. This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- AArch32 load and store multiple registers instructions increment the counter by the number of registers transferred.
- Atomic store instructions increment the counter by 1. These are instructions that atomically update a value in memory without returning a value to a register.
- Atomic load, compare and swap of a single register, and swap instructions increment the counter by 2. Atomic load instructions are instructions that atomically update a value in memory, returning a value to a register.
- Compare and swap of a pair of registers increment the counter by 4.
- Advanced SIMD LD1R instructions increment the counter by 1.
- Advanced SIMD LD[1-4] and ST[1-4] instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.
- DC ZVA and DC GZVA instructions increment by an IMPLEMENTATION DEFINED amount.

Note

When a replicating load instruction loads data of the same size as the vector length, which causes no replication to occur, the instruction is counted.

LD_ALIGN_LAT, Load with additional latency from alignment

The counter counts each ‘Memory-read operation’ counted by [LDST_ALIGN_LAT](#).

PMCEID1_EL0[33] reads as 1 if this event is implemented and 0 otherwise.

LD_FIXED_BYTES_SPEC, Non-scalable load bytes speculatively executed

The counter counts each speculatively read byte counted by [LDST_FIXED_BYTES_SPEC](#).

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred. That is, the counter is incremented by:

- Half the value that the [LDST_FIXED_BYTES_SPEC](#) event counts if the operation is a load atomic, compare and swap, or swap operation.
- The same as for [LDST_FIXED_BYTES_SPEC](#) if the operation is any other load operation.

LD_FIXED_OPS_SPEC, Non-scalable load element operations speculatively executed

The counter counts each [speculatively executed](#) ‘Memory-read operation’ counted by [LDST_FIXED_OPS_SPEC](#).

For each instruction, the counter is incremented by the number of operations specified by the instruction. That is, the counter is incremented by:

- Half the value that the [LDST_FIXED_OPS_SPEC](#) event counts if the operation is a load atomic, compare and swap, or swap operation.
- The same as for [LDST_FIXED_OPS_SPEC](#) if the operation is any other load operation.

LL_CACHE_HIT, Last level cache hit

The counter counts each access counted by [LL_CACHE](#) that hits in the Last level cache.

LL_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Last level cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LL_CACHE_HIT_LDST_RETIRED](#) that hit modified data in Last level cache.

LL_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Last level cache

The counter counts each [architecturally executed](#) instruction counted by [LL_CACHE_HITM_LDST_RETIRED](#) that is a Memory-reading instruction.

LL_CACHE_HITM_RD, Last level cache demand access hit modified, read

The counter counts each cache access counted by [LL_CACHE_HIT_RD](#) that accesses a previously modified cache location.

LL_CACHE_HITM_RW, Last level cache demand access hit modified

The counter counts each cache access counted by [LL_CACHE_HIT_RW](#) that accesses a previously modified cache location.

LL_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Last level cache

The counter counts each [architecturally executed](#) instruction counted by [LL_CACHE_HITM_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

LL_CACHE_HITM_WR, Last level cache demand access hit modified, write

The counter counts each cache access counted by [LL_CACHE_HIT_WR](#) that accesses a previously modified cache location.

LL_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Last level cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that hit in Last level cache.

LL_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Last level cache

The counter counts each [architecturally executed](#) instruction counted by [LL_CACHE_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

LL_CACHE_HIT_PRFM, Last level cache software prefetch hit

The counter counts each software prefetch counted by [LL_CACHE_PRFM](#) that hits in the Last level cache.

LL_CACHE_HIT_RD, Last level cache demand access hit, read

The counter counts each demand read counted by [LL_CACHE_RD](#) that hits in the Last level cache.

LL_CACHE_HIT_RD_FHWPRF, Last level cache demand access first hit, read, fetched by hardware prefetcher

The counter counts each first hit counted by [LL_CACHE_HIT_RW_FHWPRF](#) that is due to a demand ‘Memory-read operation’.

LL_CACHE_HIT_RD_FPRF, Last level cache demand access first hit, read, fetched by prefetch

The counter counts each first hit counted by [LL_CACHE_HIT_RW_FPRF](#) that is due to a demand ‘Memory-read operation’.

LL_CACHE_HIT_RD_FPRFM, Last level cache demand access first hit, read, fetched by software prefetch

The counter counts each first hit counted by [LL_CACHE_HIT_RW_FPRFM](#) that is due to a demand ‘Memory-read operation’.

LL_CACHE_HIT_RW, Last level cache demand access hit

The counter counts each demand access counted by [LL_CACHE_RW](#) that hits in the Last level cache.

LL_CACHE_HIT_RW_FHWPRF, Last level cache demand access first hit, fetched by hardware prefetcher

The counter counts each demand access first hit counted by [LL_CACHE_HIT_RW_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the [LL_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

LL_CACHE_HIT_RW_FPRF, Last level cache demand access first hit, fetched by prefetch

The counter counts each demand access first hit counted by [LL_CACHE_HIT_RW](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the [LL_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

LL_CACHE_HIT_RW_FPRFM, Last level cache demand access first hit, fetched by software prefetch

The counter counts each demand access first hit counted by [LL_CACHE_HIT_RW_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the [LL_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

LL_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Last level cache

The counter counts each [architecturally executed](#) instruction counted by [LL_CACHE_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

LL_CACHE_HIT_WR, Last level cache demand access hit, write

The counter counts each demand write counted by [LL_CACHE_WR](#) that hits in the Last level cache.

LL_CACHE_HIT_WR_FHWPRF, Last level cache demand access first hit, write, fetched by hardware prefetcher

The counter counts each first hit counted by [LL_CACHE_HIT_RW_FHWPRF](#) that is due to a demand [Memory-write operation](#).

LL_CACHE_HIT_WR_FPRF, Last level cache demand access first hit, write, fetched by prefetch

The counter counts each first hit counted by [LL_CACHE_HIT_RW_FPRF](#) that is due to a demand [Memory-write operation](#).

LL_CACHE_HIT_WR_FPRFM, Last level cache demand access first hit, write, fetched by software prefetch

The counter counts each first hit counted by [LL_CACHE_HIT_RW_FPRFM](#) that is due to a demand [Memory-write operation](#).

LL_CACHE_HWPRF, Last level cache hardware prefetch

The counter counts each access counted by [LL_CACHE_PRF](#) that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Last level cache.

LL_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Last level cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LDST_ANY_RETIRED](#) that missed in Last level cache.

LL_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Last level cache

The counter counts each [architecturally executed](#) instruction counted by [LL_CACHE_MISS_LDST_RETIRED](#) that is a Memory-reading instruction.

LL_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Last level cache

The counter counts each [architecturally executed](#) instruction counted by [LL_CACHE_MISS_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

LL_CACHE_PRF, Last level cache, prefetch hit

The counter counts each access counted by [LL_CACHE](#) that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Last level cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

LL_CACHE_PRFM, Last level cache software prefetch

The counter counts each access counted by [LL_CACHE_PRF](#) that is due to a prefetch instruction.

This includes accesses to the Last level cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event [LL_CACHE_RW](#) is implemented.

LL_CACHE_REFILL, Last level cache refill

The counter counts each access counted by [LL_CACHE](#) that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 to Last level caches, even if the data is ultimately not allocated into the Last level cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Last level cache that cause a refill that is satisfied by fetching data from memory, or a Last level cache of another PE.
- Refills of and write-backs from any other data, instruction, or unified caches of this PE that cause a refill from outside of the Level 1 to Last level caches of this PE.
- Accesses to the Last level cache that cause a refill of a higher level cache of this PE from outside of the Level 1 to Last level caches of this PE, even if there is no refill of the Last level cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

LL_CACHE_REFILL_HWPRF, Last level cache refill, hardware prefetch

The counter counts each hardware prefetch counted by [LL_CACHE_HWPRF](#) that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

LL_CACHE_REFILL_PRF, Last level cache refill, prefetch hit

The counter counts each access counted by [LL_CACHE_PRF](#) that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

LL_CACHE_REFILL_PRFM, Last level cache refill, software prefetch

The counter counts each access counted by [LL_CACHE_PRFM](#) that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

LL_CACHE_REFILL_WR, Last level cache refill, write

The counter counts each access counted by both [LL_CACHE_REFILL](#) and [LL_CACHE_WR](#).

That is, every refill of the Last level cache counted by [LL_CACHE_REFILL](#) that is caused by a [Memory-write operation](#).

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

LL_CACHE_RW, Last level cache demand access

The counter counts each access counted by [LL_CACHE](#) that is due to a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

This includes:

- Accesses made by [speculatively executed](#) instructions.
- Accesses to the Last level cache due to a refill of another cache caused by a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

Arm recommends that this event is implemented if any of the following are true:

- Event [LL_CACHE_PRFM](#) is implemented.
- Event [LL_CACHE_HWPRF](#) is implemented.

LL_CACHE_WB, Last level cache write-back

The counter counts each write-back of data from the Last level cache to outside of the Level 1 to Last level caches.

For example:

- A write-back of a cache line to memory.
- A write-back of a recently fetched cache line that has not been allocated to the Last level cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Last level cache to outside of Level 1 to Last level caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 to Last level caches or memory.
- Writes that write through the Last level cache to outside of the Level 1 to Last level caches.
- Transfers of data from the Last level cache to another cache to satisfy a refill of the other cache.

A write-back is attributable to one of the following, and it is IMPLEMENTATION DEFINED which:

- The PE or other agent that generated the request that caused the write-back. This might not be the same agent that caused the data to be allocated into the cache.

- If the cache is shared, the PE or other agent that caused the data to be allocated into the cache.
- If the cache is shared, the PE or other agent that last accessed the data when it was in the cache. If the data has not been accessed since being allocated, then the PE or other agent that caused the data to be allocated into the cache.

If the cache is shared, then the write-back event is counted only if it is attributable to the PE counting the event.

If the cache is not shared, then the event is counted by the PE that the cache is attached to, even if the write-back is not attributable to that PE. For example, a requestor outside of the PE made a coherency request that resulted in write-back.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a DC ZVA operation.

LL_CACHE_WR, Last level cache access, write

If the [LL_CACHE_RW](#) event is implemented, the counter counts each access counted by [LL_CACHE_RW](#) that is a [Memory-write operation](#).

If the [LL_CACHE_RW](#) event is not implemented, the counter counts each access counted by [LL_CACHE](#) that is a [Memory-write operation](#).

If the cache is shared and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of `PMEVTYPER<n>_EL0.MT` for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of `PMEVTYPER<n>_EL0.MT`.

LL_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Last level cache

The counter counts each [architecturally executed](#) Memory-reading instruction or [Memory-writing instruction](#) counted by [LL_CACHE_HIT_LDST_RETIRED](#) that hit a recently fetched line in Last level cache.

LL_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Last level cache

The counter counts each [architecturally executed](#) instruction counted by [LL_LFB_HIT_LDST_RETIRED](#) that is a Memory-reading instruction.

LL_LFB_HIT_RD, Last level cache demand access line-fill buffer hit, read

The counter counts each demand access counted by [LL_CACHE_HIT_RD](#) that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

LL_LFB_HIT_RD_FHWPRF, Last level cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by [LL_LFB_HIT_RW_FHWPRF](#) that is due to a demand 'Memory-read operation'.

LL_LFB_HIT_RD_FPRF, Last level cache demand access line-fill buffer first hit, read, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by [LL_LFB_HIT_RW_FPRF](#) that is due to a demand 'Memory-read operation'.

LL_LFB_HIT_RD_FPRFM, Last level cache demand access line-fill buffer first hit, read, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by [LL_LFB_HIT_RW_FPRFM](#) that is due to a demand ‘Memory-read operation’.

LL_LFB_HIT_RW, Last level cache demand access line-fill buffer hit

The counter counts each demand access counted by [LL_CACHE_HIT_RW](#) that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

LL_LFB_HIT_RW_FHWPRF, Last level cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher

The counter counts each demand access line-fill buffer first hit counted by [LL_LFB_HIT_RW_FPRF](#) where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [LL_CACHE_REFILL_HWPRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

LL_LFB_HIT_RW_FPRF, Last level cache demand access line-fill buffer first hit, recently fetched by prefetch

The counter counts each demand access line-fill buffer first hit counted by [LL_LFB_HIT_RW](#) where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [LL_CACHE_REFILL_PRF](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

LL_LFB_HIT_RW_FPRFM, Last level cache demand access line-fill buffer first hit, recently fetched by software prefetch

The counter counts each demand access line-fill buffer first hit counted by [LL_LFB_HIT_RW_FPRF](#) where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the [LL_CACHE_REFILL_PRFM](#) event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

LL_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Last level cache

The counter counts each [architecturally executed](#) instruction counted by [LL_LFB_HIT_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

LL_LFB_HIT_WR, Last level cache demand access line-fill buffer hit, write

The counter counts each demand access counted by [LL_CACHE_HIT_WR](#) that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

LL_LFB_HIT_WR_FHWPRF, Last level cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher

The counter counts each line-fill buffer first hit counted by [LL_LFB_HIT_RW_FHWPRF](#) that is due to a demand [Memory-write operation](#).

LL_LFB_HIT_WR_FPRF, Last level cache demand access line-fill buffer first hit, write, recently fetched by prefetch

The counter counts each line-fill buffer first hit counted by [LL_LFB_HIT_RW_FPRF](#) that is due to a demand [Memory-write operation](#).

LL_LFB_HIT_WR_FPRFM, Last level cache demand access line-fill buffer first hit, write, recently fetched by software prefetch

The counter counts each line-fill buffer first hit counted by [LL_LFB_HIT_RW_FPRFM](#) that is due to a demand [Memory-write operation](#).

LOCAL_MEM, Access to memory attached to this device

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) access to external memory attached to this device.

In a multi-socket system this means attached to the socket that contains the PE. For more information, see [REMOTE_ACCESS](#).

In systems where there multiple types of memory attached to this device with different performance characteristics, it is IMPLEMENTATION DEFINED whether accesses to all external memory types are counted by this event, or are classified as remote accesses and counted by [REMOTE_MEM](#).

For example, in a system with an expansion memory connected to the device that has significantly higher latency than the main system memory, accesses to the main system memory might be counted by this event and accesses to the expansion memory by [REMOTE_MEM](#).

LOCAL_MEM_LDST_RETIRED, Load or store instruction architecturally executed, access to memory attached to this device

The counter counts each Memory-reading instruction or [Memory-writing instruction architecturally executed](#) counted by [LDST_ANY_RETIRED](#) that accessed external memory attached to this device.

LOCAL_MEM_LD_RETIRED, Load instruction architecturally executed, access to memory attached to this device

The counter counts each [architecturally executed](#) instruction counted by [LOCAL_MEM_LDST_RETIRED](#) that is a Memory-reading instruction.

LOCAL_MEM_PRFM, Access to memory attached to this device, software prefetch

The counter counts each access counted by [LOCAL_MEM](#) that is due to a prefetch instruction.

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by [LOCAL_MEM_RW](#) or [LOCAL_MEM_PRFM](#).

LOCAL_MEM_RD, Access to memory attached to this device, demand access, read

The counter counts each access counted by [LOCAL_MEM_RW](#) that is a demand ‘Memory-read operation’.

LOCAL_MEM_RW, Access to memory attached to this device, demand access

The counter counts each access counted by [LOCAL_MEM](#) that is a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by [LOCAL_MEM_RW](#) or [LOCAL_MEM_PRFM](#).

LOCAL_MEM_ST_RETIRED, Store instruction architecturally executed, access to memory attached to this device

The counter counts each [architecturally executed](#) instruction counted by [LOCAL_MEM_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

LOCAL_MEM_WR, Access to memory attached to this device, demand access, write

The counter counts each access counted by [LOCAL_MEM_RW](#) that is a demand [Memory-write operation](#).

MEM_ACCESS_PRFM, Data memory access, software prefetch

The counter counts each access counted by [MEM_ACCESS](#) that is due to a prefetch instruction.

MEM_ACCESS_RD_PERCYC, Data memory reads in progress

The counter increments by the number of data memory reads counted by [MEM_ACCESS_RD](#) in progress on each [Processor cycle](#).

The ratio $\text{MEM_ACCESS_RD_PERCYC} \div \text{MEM_ACCESS_RD}$ is the mean duration of data memory reads in [Processor cycles](#).

MEM_ACCESS_RW, Data memory access, demand access

The counter counts each access counted by [MEM_ACCESS](#) that is a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

This includes accesses made by speculative instructions.

N1_CACHE1_HITM_RD, Cache type 1 at distance 1 demand access hit modified, read

The counter counts each demand read counted by [N1_CACHE_HITM_RD](#) that accesses a previously modified location in a first cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE1_HIT_PRFM, Cache type 1 at distance 1 software prefetch hit

The counter counts each software prefetch counted by [N1_CACHE_HIT_PRFM](#) due to a software prefetch accessing a first cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE1_HIT_RD, Cache type 1 at distance 1 demand access hit, read

The counter counts each demand read counted by [N1_CACHE_HIT_RD](#) due to a demand read accessing a first cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE2_HITM_RD, Cache type 2 at distance 1 demand access hit modified, read

The counter counts each demand read counted by [N1_CACHE_HITM_RD](#) that accesses a previously modified location in a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE2_HIT_PRFM, Cache type 2 at distance 1 software prefetch hit

The counter counts each software prefetch counted by [N1_CACHE_HIT_PRFM](#) due to a software prefetch accessing a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE2_HIT_RD, Cache type 2 at distance 1 demand access hit, read

The counter counts each demand read counted by [N1_CACHE_HIT_RD](#) due to a demand read accessing a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE3_HITM_RD, Cache type 3 at distance 1 demand access hit modified, read

The counter counts each demand read counted by [N1_CACHE_HITM_RD](#) that accesses a previously modified location in a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE3_HIT_PRFM, Cache type 3 at distance 1 software prefetch hit

The counter counts each software prefetch counted by [N1_CACHE_HIT_PRFM](#) due to a software prefetch accessing a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE3_HIT_RD, Cache type 3 at distance 1 demand access hit, read

The counter counts each demand read counted by [N1_CACHE_HIT_RD](#) due to a demand read accessing a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_CACHE_HITM_RD, Cache at distance 1, demand access hit modified, read

The counter counts each demand read counted by [N1_CACHE_HIT_RD](#) that accesses a previously modified location in a cache at distance 1.

N1_CACHE_HIT_PRFM, Cache at distance 1, software prefetch hit

The counter counts each cache hit at distance 1 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N1_CACHE_HIT_RD, Cache at distance 1, demand access hit, read

The counter counts each cache hit at distance 1 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N1_LFB1_HIT_RD, Cache type 1 at distance 1 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N1_LFB_HIT_RD](#) that hits a recently fetched line in a first cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_LFB2_HIT_RD, Cache type 2 at distance 1 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N1_LFB_HIT_RD](#) that hits a recently fetched line in a second cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_LFB3_HIT_RD, Cache type 3 at distance 1 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N1_LFB_HIT_RD](#) that hits a recently fetched line in a third cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_LFB_HIT_RD, Cache at distance 1 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N1_CACHE_HIT_RD](#) that hits a recently fetched line in a cache at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

N1_MEM1_PRFM, Access to type 1 memory at distance 1, software prefetch

The counter counts each software prefetch counted by [N1_MEM_PRFM](#) to a first memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_MEM1_RD, Access to type 1 memory at distance 1, demand access, read

The counter counts each demand read counted by [N1_MEM_RD](#) to a first memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM1_RD](#).

For example, the first memory type might be commodity DRAM connected to a device. However, this must be the same for all devices in the system, such that, for example, [N1_MEM1_RD](#) refers to commodity DRAM for a PE at distance 1

from the memory, regardless of which device the PE is located on, and [N4_MEM1_RD](#) refers to the same memory for any PE located at distance 4. This means that when a process counting the [N1_MEM1_RD](#) event migrates between PEs on different devices, its relative view of the first memory type remains unchanged, even if the distance changes.

N1_MEM2_PRFM, Access to type 2 memory at distance 1, software prefetch

The counter counts each software prefetch counted by [N1_MEM_PRFM](#) to a second memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_MEM2_RD, Access to type 2 memory at distance 1, demand access, read

The counter counts each demand read counted by [N1_MEM_RD](#) to a second memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM2_RD](#).

For example, the second memory type might be bulk expansion memory connected to a device. However, this must be the same for all devices in the system, such that, for example, [N1_MEM2_RD](#) refers to bulk expansion memory for a PE at distance 1 from the memory, regardless of which device the PE is located on, and [N4_MEM2_RD](#) refers to the same memory for any PE located at distance 4. This means that when a process counting the [N1_MEM2_RD](#) event migrates between PEs on different devices, its relative view of the second memory type remains unchanged, even if the distance changes.

N1_MEM3_PRFM, Access to type 3 memory at distance 1, software prefetch

The counter counts each software prefetch counted by [N1_MEM_PRFM](#) to a third memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N1_MEM3_RD, Access to type 3 memory at distance 1, demand access, read

The counter counts each demand read counted by [N1_MEM_RD](#) to a third memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM3_RD](#).

For example, the third memory type might be high-bandwidth memory connected to a device. However, this must be the same for all devices in the system, such that, for example, [N1_MEM3_RD](#) refers to high-bandwidth memory for a PE at distance 1 from the memory, regardless of which device the PE is located on, and [N4_MEM3_RD](#) refers to the same memory for any PE located at distance 4. This means that when a process counting the [N1_MEM3_RD](#) event migrates between PEs on different devices, its relative view of the third memory type remains unchanged, even if the distance changes.

N1_MEM_PRFM, Access to memory at distance 1, software prefetch

The counter counts each software prefetch to memory at distance 1.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N1_MEM_RD, Access to memory at distance 1, demand access, read

The counter counts each demand read to memory at distance 1.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N2_CACHE1_HITM_RD, Cache type 1 at distance 2 demand access hit modified, read

The counter counts each demand read counted by [N2_CACHE_HITM_RD](#) that accesses a previously modified location in a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE1_HIT_PRFM, Cache type 1 at distance 2 software prefetch hit

The counter counts each software prefetch counted by [N2_CACHE_HIT_PRFM](#) due to a software prefetch accessing a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE1_HIT_RD, Cache type 1 at distance 2 demand access hit, read

The counter counts each demand read counted by [N2_CACHE_HIT_RD](#) due to a demand read accessing a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE2_HITM_RD, Cache type 2 at distance 2 demand access hit modified, read

The counter counts each demand read counted by [N2_CACHE_HITM_RD](#) that accesses a previously modified location in a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE2_HIT_PRFM, Cache type 2 at distance 2 software prefetch hit

The counter counts each software prefetch counted by [N2_CACHE_HIT_PRFM](#) due to a software prefetch accessing a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE2_HIT_RD, Cache type 2 at distance 2 demand access hit, read

The counter counts each demand read counted by [N2_CACHE_HIT_RD](#) due to a demand read accessing a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE3_HITM_RD, Cache type 3 at distance 2 demand access hit modified, read

The counter counts each demand read counted by [N2_CACHE_HITM_RD](#) that accesses a previously modified location in a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE3_HIT_PRFM, Cache type 3 at distance 2 software prefetch hit

The counter counts each software prefetch counted by [N2_CACHE_HIT_PRFM](#) due to a software prefetch accessing a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE3_HIT_RD, Cache type 3 at distance 2 demand access hit, read

The counter counts each demand read counted by [N2_CACHE_HIT_RD](#) due to a demand read accessing a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_CACHE_HITM_RD, Cache at distance 2, demand access hit modified, read

The counter counts each demand read counted by [N2_CACHE_HIT_RD](#) that accesses a previously modified location in a cache at distance 2.

N2_CACHE_HIT_PRFM, Cache at distance 2, software prefetch hit

The counter counts each cache hit at distance 2 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 2 represents a further distance of travel than cache distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N2_CACHE_HIT_RD, Cache at distance 2, demand access hit, read

The counter counts each cache hit at distance 2 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 2 represents a further distance of travel than cache distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N2_LFB1_HIT_RD, Cache type 1 at distance 2 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N2_LFB_HIT_RD](#) that hits a recently fetched line in a first cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_LFB2_HIT_RD, Cache type 2 at distance 2 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N2_LFB_HIT_RD](#) that hits a recently fetched line in a second cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_LFB3_HIT_RD, Cache type 3 at distance 2 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N2_LFB_HIT_RD](#) that hits a recently fetched line in a third cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_LFB_HIT_RD, Cache at distance 2 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N2_CACHE_HIT_RD](#) that hits a recently fetched line in a cache at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

N2_MEM1_PRFM, Access to type 1 memory at distance 2, software prefetch

The counter counts each software prefetch counted by [N2_MEM_PRFM](#) to a first memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_MEM1_RD, Access to type 1 memory at distance 2, demand access, read

The counter counts each demand read counted by [N2_MEM_RD](#) to a first memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM1_RD](#).

N2_MEM2_PRFM, Access to type 2 memory at distance 2, software prefetch

The counter counts each software prefetch counted by [N2_MEM_PRFM](#) to a second memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_MEM2_RD, Access to type 2 memory at distance 2, demand access, read

The counter counts each demand read counted by [N2_MEM_RD](#) to a second memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM2_RD](#).

N2_MEM3_PRFM, Access to type 3 memory at distance 2, software prefetch

The counter counts each software prefetch counted by [N2_MEM_PRFM](#) to a third memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N2_MEM3_RD, Access to type 3 memory at distance 2, demand access, read

The counter counts each demand read counted by [N2_MEM_RD](#) to a third memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM3_RD](#).

N2_MEM_PRFM, Access to memory at distance 2, software prefetch

The counter counts each software prefetch to memory at distance 2.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 2 represents a further distance of travel than distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N2_MEM_RD, Access to memory at distance 2, demand access, read

The counter counts each demand read to memory at distance 2.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 2 represents a further distance of travel than distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N3_CACHE1_HITM_RD, Cache type 1 at distance 3 demand access hit modified, read

The counter counts each demand read counted by [N3_CACHE_HITM_RD](#) that accesses a previously modified location in a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE1_HIT_PRFM, Cache type 1 at distance 3 software prefetch hit

The counter counts each software prefetch counted by [N3_CACHE_HIT_PRFM](#) due to a software prefetch accessing a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE1_HIT_RD, Cache type 1 at distance 3 demand access hit, read

The counter counts each demand read counted by [N3_CACHE_HIT_RD](#) due to a demand read accessing a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE2_HITM_RD, Cache type 2 at distance 3 demand access hit modified, read

The counter counts each demand read counted by [N3_CACHE_HITM_RD](#) that accesses a previously modified location in a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE2_HIT_PRFM, Cache type 2 at distance 3 software prefetch hit

The counter counts each software prefetch counted by [N3_CACHE_HIT_PRFM](#) due to a software prefetch accessing a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE2_HIT_RD, Cache type 2 at distance 3 demand access hit, read

The counter counts each demand read counted by [N3_CACHE_HIT_RD](#) due to a demand read accessing a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE3_HITM_RD, Cache type 3 at distance 3 demand access hit modified, read

The counter counts each demand read counted by [N3_CACHE_HITM_RD](#) that accesses a previously modified location in a third cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE3_HIT_PRFM, Cache type 3 at distance 3 software prefetch hit

The counter counts each software prefetch counted by [N3_CACHE_HIT_PRFM](#) due to a software prefetch accessing a third cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE3_HIT_RD, Cache type 3 at distance 3 demand access hit, read

The counter counts each demand read counted by [N3_CACHE_HIT_RD](#) due to a demand read accessing a third cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_CACHE_HITM_RD, Cache at distance 3, demand access hit modified, read

The counter counts each demand read counted by [N3_CACHE_HIT_RD](#) that accesses a previously modified location in a cache at distance 3.

N3_CACHE_HIT_PRFM, Cache at distance 3, software prefetch hit

The counter counts each cache hit at distance 3 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 3 represents a further distance of travel than cache distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N3_CACHE_HIT_RD, Cache at distance 3, demand access hit, read

The counter counts each cache hit at distance 3 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 3 represents a further distance of travel than cache distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N3_LFB1_HIT_RD, Cache type 1 at distance 3 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N3_LFB_HIT_RD](#) that hits a recently fetched line in a first cache type at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_LFB2_HIT_RD, Cache type 2 at distance 3 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N3_LFB_HIT_RD](#) that hits a recently fetched line in a second cache type at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_LFB3_HIT_RD, Cache type 3 at distance 3 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N3_LFB_HIT_RD](#) that hits a recently fetched line in a third cache type at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_LFB_HIT_RD, Cache at distance 3 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N3_CACHE_HIT_RD](#) that hits a recently fetched line in a cache at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

N3_MEM1_PRFM, Access to type 1 memory at distance 3, software prefetch

The counter counts each software prefetch counted by [N3_MEM_PRFM](#) to a first memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_MEM1_RD, Access to type 1 memory at distance 3, demand access, read

The counter counts each demand read counted by [N3_MEM_RD](#) to a first memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM1_RD](#).

N3_MEM2_PRFM, Access to type 2 memory at distance 3, software prefetch

The counter counts each software prefetch counted by [N3_MEM_PRFM](#) to a second memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_MEM2_RD, Access to type 2 memory at distance 3, demand access, read

The counter counts each demand read counted by [N3_MEM_RD](#) to a second memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM2_RD](#).

N3_MEM3_PRFM, Access to type 3 memory at distance 3, software prefetch

The counter counts each software prefetch counted by [N3_MEM_PRFM](#) to a third memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N3_MEM3_RD, Access to type 3 memory at distance 3, demand access, read

The counter counts each demand read counted by [N3_MEM_RD](#) to a third memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM3_RD](#).

N3_MEM_PRFM, Access to memory at distance 3, software prefetch

The counter counts each software prefetch to memory at distance 3.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 3 represents a further distance of travel than distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N3_MEM_RD, Access to memory at distance 3, demand access, read

The counter counts each demand read to memory at distance 3.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 3 represents a further distance of travel than distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N4_CACHE1_HITM_RD, Cache type 1 at distance 4 demand access hit modified, read

The counter counts each demand read counted by [N4_CACHE_HITM_RD](#) that accesses a previously modified location in a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE1_HIT_PRFM, Cache type 1 at distance 4 software prefetch hit

The counter counts each software prefetch counted by [N4_CACHE_HIT_PRFM](#) due to a software prefetch accessing a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE1_HIT_RD, Cache type 1 at distance 4 demand access hit, read

The counter counts each demand read counted by [N4_CACHE_HIT_RD](#) due to a demand read accessing a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE2_HITM_RD, Cache type 2 at distance 4 demand access hit modified, read

The counter counts each demand read counted by [N4_CACHE_HITM_RD](#) that accesses a previously modified location in a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE2_HIT_PRFM, Cache type 2 at distance 4 software prefetch hit

The counter counts each software prefetch counted by [N4_CACHE_HIT_PRFM](#) due to a software prefetch accessing a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE2_HIT_RD, Cache type 2 at distance 4 demand access hit, read

The counter counts each demand read counted by [N4_CACHE_HIT_RD](#) due to a demand read accessing a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE3_HITM_RD, Cache type 3 at distance 4 demand access hit modified, read

The counter counts each demand read counted by [N4_CACHE_HITM_RD](#) that accesses a previously modified location in a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE3_HIT_PRFM, Cache type 3 at distance 4 software prefetch hit

The counter counts each software prefetch counted by [N4_CACHE_HIT_PRFM](#) due to a software prefetch accessing a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE3_HIT_RD, Cache type 3 at distance 4 demand access hit, read

The counter counts each demand read counted by [N4_CACHE_HIT_RD](#) due to a demand read accessing a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_CACHE_HITM_RD, Cache at distance 4, demand access hit modified, read

The counter counts each demand read counted by [N4_CACHE_HIT_RD](#) that accesses a previously modified location in a cache at distance 4.

N4_CACHE_HIT_PRFM, Cache at distance 4, software prefetch hit

The counter counts each cache hit at distance 4 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 4 represents a further distance of travel than cache distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N4_CACHE_HIT_RD, Cache at distance 4, demand access hit, read

The counter counts each cache hit at distance 4 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 4 represents a further distance of travel than cache distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N4_LFB1_HIT_RD, Cache type 1 at distance 4 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N4_LFB_HIT_RD](#) that hits a recently fetched line in a first cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_LFB2_HIT_RD, Cache type 2 at distance 4 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N4_LFB_HIT_RD](#) that hits a recently fetched line in a second cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_LFB3_HIT_RD, Cache type 3 at distance 4 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N4_LFB_HIT_RD](#) that hits a recently fetched line in a third cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_LFB_HIT_RD, Cache at distance 4 demand access line-fill buffer hit, read

The counter counts each demand read counted by [N4_CACHE_HIT_RD](#) that hits a recently fetched line in a cache at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

N4_MEM1_PRFM, Access to type 1 memory at distance 4, software prefetch

The counter counts each software prefetch counted by [N4_MEM_PRFM](#) to a first memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_MEM1_RD, Access to type 1 memory at distance 4, demand access, read

The counter counts each demand read counted by [N4_MEM_RD](#) to a first memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM1_RD](#).

N4_MEM2_PRFM, Access to type 2 memory at distance 4, software prefetch

The counter counts each software prefetch counted by [N4_MEM_PRFM](#) to a second memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_MEM2_RD, Access to type 2 memory at distance 4, demand access, read

The counter counts each demand read counted by [N4_MEM_RD](#) to a second memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM2_RD](#).

N4_MEM3_PRFM, Access to type 3 memory at distance 4, software prefetch

The counter counts each software prefetch counted by [N4_MEM_PRFM](#) to a third memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

N4_MEM3_RD, Access to type 3 memory at distance 4, demand access, read

The counter counts each demand read counted by [N4_MEM_RD](#) to a third memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also [N1_MEM3_RD](#).

N4_MEM_PRFM, Access to memory at distance 4, software prefetch

The counter counts each software prefetch to memory at distance 4.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 4 represents a further distance of travel than distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

N4_MEM_RD, Access to memory at distance 4, demand access, read

The counter counts each demand read to memory at distance 4.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 4 represents a further distance of travel than distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

NONFP_SPEC, Non-floating-point operation speculatively executed

The counter counts each [speculatively executed](#) data-processing operation due to any of:

- A scalar instruction that would be counted by the [DP_SPEC](#) event.
- An A64 Advanced SIMD data processing instruction defined in the section Data processing - SIMD and floating-point that would not be counted by [FP_SPEC](#).
- An SVE instruction with vector source or destination registers that would not be counted by [FP_SPEC](#).

It is IMPLEMENTATION DEFINED which non-floating-point data-processing operations are counted in AArch32 state.

PMU_SNAPSHOT, Successful PMU capture event

The counter counts each PMU snapshot Capture event that was successful, that is, PMSSCR_EL1.NC is set to 0.

It is CONSTRAINED UNPREDICTABLE whether the counter counts successful Capture events when the PE is in Debug state.

PRF_SPEC, Operation speculatively executed, prefetch

The counter counts each [speculatively executed](#) prefetch operation due to any of the following A64 instructions:

- Scalar: PRFM, PRFUM, or RPRFM.

It is IMPLEMENTATION DEFINED which prefetch operations are counted in AArch32 state.

REMOTE_ACCESS_WR, Access to a remote device, write

The counter counts each access counted by [REMOTE_ACCESS](#) that is a [Memory-write operation](#).

REMOTE_CACHE_HITM_RD, Remote cache demand access hit modified, read

The counter counts each cache hit on a remote device counted by [REMOTE_CACHE_HIT_RD](#) that accesses a previously modified location.

REMOTE_CACHE_HIT_PRFM, Remote cache software prefetch hit

The counter counts each cache hit on a remote device due to a software prefetch.

REMOTE_CACHE_HIT_RD, Remote cache demand access hit, read

The counter counts each cache hit on a remote device due to a demand read.

REMOTE_LFB_HIT_RD, Remote cache demand access line-fill buffer hit, read

The counter counts each cache hit on a remote device counted by [REMOTE_CACHE_HIT_RD](#) that hits a recently fetched line.

That is, the read hits a cache line that is in the process of being loaded into the cache attached to a remote device, and so does not generate a new refill, but has to wait for the previous refill to complete.

REMOTE_MEM, Access to memory attached to a remote device

The counter counts each ‘Memory-read operation’ or [Memory-write operation](#) access counted by [REMOTE_ACCESS](#) to external memory attached to a remote device.

The counter also counts any accesses to external memory that are not counted by [LOCAL_MEM](#) because the implementation classifies them as remote accesses. For example, accesses to expansion memory connected to the device that has significantly higher latency than the main system memory might be classified as remote accesses. For more information, see [LOCAL_MEM](#).

REMOTE_MEM_LDST_RETIRED, Load or store instruction architecturally executed, access to memory attached to a remote device

The counter counts each Memory-reading instruction or [Memory-writing instruction architecturally executed](#) counted by [LDST_ANY_RETIRED](#) that accessed external memory attached to a remote device.

REMOTE_MEM_LD_RETIRED, Load instruction architecturally executed, access to memory attached to a remote device

The counter counts each [architecturally executed](#) instruction counted by [REMOTE_MEM_LDST_RETIRED](#) that is a Memory-reading instruction.

REMOTE_MEM_PRFM, Access to memory attached to a remote device, software prefetch

The counter counts each access counted by [REMOTE_MEM](#) that is due to a prefetch instruction.

REMOTE_MEM_RD, Access to memory attached to a remote device, demand access, read

The counter counts each access counted by [REMOTE_MEM_RW](#) that is a demand ‘Memory-read operation’.

REMOTE_MEM_RW, Access to memory attached to a remote device, demand access

The counter counts each access counted by [REMOTE_MEM](#) that is a demand ‘Memory-read operation’ or demand [Memory-write operation](#).

REMOTE_MEM_ST_RETIRED, Store instruction architecturally executed, access to memory attached to a remote device

The counter counts each [architecturally executed](#) instruction counted by [REMOTE_MEM_LDST_RETIRED](#) that is a [Memory-writing instruction](#).

REMOTE_MEM_WR, Access to memory attached to a remote device, demand access, write

The counter counts each access counted by [REMOTE_MEM_RW](#) that is a demand [Memory-write operation](#).

SAMPLE_BUFFER_FULL, Profiling Buffer full

The counter counts each Profiling Buffer full management event.

SAMPLE_COLLISION, Statistical Profiling sample collided with previous sample

The counter counts each time the sample interval counter reaches zero and is reloaded, and the sample collides with a previous sample.

For more information, see ‘Sample collisions’.

PMCEID0_EL0[35] reads as 1 if this event is implemented and 0 otherwise.

SAMPLE_FEED, Statistical Profiling sample taken

The counter counts each time the sample interval counter reaches zero and is reloaded, and the sample does not collide with the previous sample.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

PMCEID0_EL0[33] reads as 1 if this event is implemented and 0 otherwise.

SAMPLE_FEED_DS, Statistical Profiling sample taken, selected Data Source

The counter counts each SPE sample that is a load operations where PMSDSFR_EL1[s] is 1 and s is bits [5:0] of the sampled Data Source.

The values of PMSFCR_EL1.FDS are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

SAMPLE_FILTRATE, Statistical Profiling sample taken and not removed by filtering

The counter counts each sample counted by [SAMPLE_FEED](#) that is not removed by filtering.

Sample records that are not removed by filtering, but are discarded before being written to the Profiling Buffer because of a Profiling Buffer management event, are counted.

PMCEID0_EL0[34] reads as 1 if this event is implemented and 0 otherwise.

SAMPLE_POP, Statistical Profiling sample population

The counter counts each operation that might be sampled, whether or not the operation was sampled.

The counter does not count when profiling is disabled.

PMCEID0_EL0[32] reads as 1 if this event is implemented and 0 otherwise.

SE_FP_ADDSUB_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point addition or subtraction

The counter counts each [speculatively executed](#) floating-point addition or subtraction operation counted by [SE_FP_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_FP_ADDSUB_SPEC](#), [SVE_FP_ADDSUB_SPEC](#), or [SME_FP_ADDSUB_SPEC](#).

SE_FP_BF16_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by [FP_BF16_MIN_SPEC](#) due to an instruction where the smallest type was BFloat16 floating-point.

SE_FP_BF16_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD BFloat16 floating-point, data-processing

The counter counts each [speculatively executed](#) BFloat16 floating-point data-processing operation counted by [SE_FP_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [SVE_FP_BF16_SPEC](#), or [SME_FP_BF16_SPEC](#).

SE_FP_DOT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point dot product

The counter counts each [speculatively executed](#) floating-point dot product operation counted by [SE_FP_SPEC](#) due to any of the following A64 instructions:

That is, each operation counted by [ASE_FP_DOT_SPEC](#), [SVE_FP_DOT_SPEC](#), or [SME_FP_DOT_SPEC](#).

SE_FP_DP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD double-precision floating-point, data-processing

The counter counts each [speculatively executed](#) double-precision floating-point data-processing operation counted by [SE_FP_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_FP_DP_SPEC](#), [SVE_FP_DP_SPEC](#), or [SME_FP_DP_SPEC](#).

It is IMPLEMENTATION DEFINED which double-precision floating-point data-processing operations are counted in AArch32 state.

SE_FP_FMA_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point multiply-add or multiply-subtract

The counter counts each [speculatively executed](#) floating-point multiply-add, multiply-add long, multiply-subtract, or multiply-subtract long operation counted by [SE_FP_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_FP_FMA_SPEC](#), [SVE_FP_FMA_SPEC](#), or [SME_FP_FMA_SPEC](#).

SE_FP_FP8_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by [FP_FP8_MIN_SPEC](#) due to an instruction where the smallest type was 8-bit floating-point.

SE_FP_HP_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by [FP_HP_MIN_SPEC](#) due to an instruction where the smallest type was half-precision floating-point.

SE_FP_HP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD half-precision floating-point, data-processing

The counter counts each [speculatively executed](#) half-precision floating-point data-processing operation counted by [SE_FP_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_FP_HP_SPEC](#), [SVE_FP_HP_SPEC](#), or [SME_FP_HP_SPEC](#).

It is IMPLEMENTATION DEFINED which half-precision floating-point data-processing operations are counted in AArch32 state.

SE_FP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point, data-processing

The counter counts each [speculatively executed](#) floating-point data-processing operation counted by [FP_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_FP_SPEC](#), [SVE_FP_SPEC](#), or [SME_FP_SPEC](#).

It is IMPLEMENTATION DEFINED which floating-point data-processing operations are counted in AArch32 state.

SE_FP_SP_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by [FP_SP_MIN_SPEC](#) due to an instruction where the smallest type was single-precision floating-point.

SE_FP_SP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD single-precision floating-point, data-processing

The counter counts each [speculatively executed](#) single-precision floating-point data-processing operation counted by [SE_FP_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_FP_SP_SPEC](#), [SVE_FP_SP_SPEC](#), or [SME_FP_SP_SPEC](#).

It is IMPLEMENTATION DEFINED which single-precision floating-point data-processing operations are counted in AArch32 state.

SE_INST_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME

The counter counts each [speculatively executed](#) operation counted by [INST_SPEC](#) that is classified as an Advanced SIMD, SVE, or SME operation.

See [ASE_SVE_INST_SPEC](#) and [SME_INST_SPEC](#) for these classifications.

SE_INT16_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 16-bit integer, data-processing

The counter counts each [speculatively executed](#) 16-bit integer data-processing operation counted by [SE_INT_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_INT16_SPEC](#), [SVE_INT16_SPEC](#), or [SME_INT16_SPEC](#).

It is IMPLEMENTATION DEFINED which 16-bit integer data-processing operations are counted in AArch32 state.

SE_INT32_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 32-bit integer, data-processing

The counter counts each [speculatively executed](#) 32-bit integer data-processing operation counted by [SE_INT_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_INT32_SPEC](#), [SVE_INT32_SPEC](#), or [SME_INT32_SPEC](#).

It is IMPLEMENTATION DEFINED which 32-bit integer data-processing operations are counted in AArch32 state.

SE_INT64_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 64-bit integer, data-processing

The counter counts each [speculatively executed](#) 64-bit integer data-processing operation counted by [SE_INT_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_INT64_SPEC](#), [SVE_INT64_SPEC](#), or [SME_INT64_SPEC](#).

It is IMPLEMENTATION DEFINED which 64-bit integer data-processing operations are counted in AArch32 state.

SE_INT8_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 8-bit integer, data-processing

The counter counts each [speculatively executed](#) 8-bit integer data-processing operation counted by [SE_INT_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_INT8_SPEC](#), [SVE_INT8_SPEC](#), or [SME_INT8_SPEC](#).

It is IMPLEMENTATION DEFINED which 8-bit integer data-processing operations are counted in AArch32 state.

SE_INT_DOT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer dot product

The counter counts each [speculatively executed](#) integer dot product operation counted by [SE_INT_SPEC](#) due to any of the following A64 instructions:

That is, each operation counted by [ASE_INT_DOT_SPEC](#), [SVE_INT_DOT_SPEC](#), or [SME_INT_DOT_SPEC](#).

SE_INT_MUL_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer multiply or multiply-accumulate

The counter counts each [speculatively executed](#) integer multiply, multiply-add, or multiply-subtract operation counted by [SE_INT_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_INT_MUL_SPEC](#), [SVE_INT_MUL_SPEC](#), or [SME_INT_MUL_SPEC](#).

SE_INT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer, data-processing

The counter counts each [speculatively executed](#) integer data-processing operation counted by [INT_SPEC](#) due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by [ASE_INT_SPEC](#), [SVE_INT_SPEC](#), or [SME_INT_SPEC](#).

It is IMPLEMENTATION DEFINED which integer data-processing operations are counted in AArch32 state.

SE_LUT_SPEC, Lookup table operation speculatively executed, Advanced SIMD, SVE, or SME data processing

The counter counts each [speculatively executed](#) LUT operation counted by [SE_INST_SPEC](#) that returns a value from a lookup table made up of a ZT register, Z or V register which is indexed by values from a Z or V register.

SE_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME data processing

The counter counts each operation counted by [INST_SPEC](#) that is an Advanced SIMD, SVE, or SME data-processing operation.

See [ASE_SVE_SPEC](#) and [SME_SPEC](#) for these classifications.

SIMD_INST_SPEC, Operation speculatively executed, SIMD

The counter counts each [speculatively executed](#) operation due to any of:

- An Advanced SIMD data-processing operation. Advanced SIMD scalar data-processing operations are not counted.
- An SVE SIMD data-processing operation. SVE non-SIMD data-processing operations are not counted.
- A structure load/store of one or more SIMD&FP registers.
- A load and replicate to one or more SIMD&FP registers.
- A scalar load/store of a SIMD&FP Q register or pair of Q registers.
- An SVE load/store.

It is IMPLEMENTATION DEFINED which Advanced SIMD operations are counted in AArch32 state.

It is IMPLEMENTATION DEFINED whether scalar loads and stores to SIMD&FP registers other than those listed above are counted.

SIMD_UOP_SPEC, Microarchitectural operation speculatively executed, SIMD

The counter counts each [speculatively executed](#) microarchitectural operation counted by [UOP_SPEC](#) due to any of:

- An SVE instruction other than non-SIMD SVE instructions.
- An A64 Advanced SIMD instruction other than an Advanced SIMD scalar instruction.

It is IMPLEMENTATION DEFINED which Advanced SIMD microarchitectural operations are counted in AArch32 state.

SME_FP_BF16_MIN_SPEC, SME data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each SME data-processing operation counted by [SE_FP_BF16_MIN_SPEC](#) due to an instruction where the smallest type was BFloat16 floating-point.

SME_FP_FP8_MIN_SPEC, SME data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each SME data-processing operation counted by [SE_FP_FP8_MIN_SPEC](#) due to an instruction where the smallest type was 8-bit floating-point.

SME_FP_HP_MIN_SPEC, SME data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each SME data-processing operation counted by [SE_FP_HP_MIN_SPEC](#) due to an instruction where the smallest type was half-precision floating-point.

SME_FP_SP_MIN_SPEC, SME data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each SME data-processing operation counted by [SE_FP_SP_MIN_SPEC](#) due to an instruction where the smallest type was single-precision floating-point.

STALL_BACKEND_BUSY, Backend stall cycles, backend busy

The counter counts each cycle counted by [STALL_BACKEND](#) when operations are available from the frontend but the backend is not able to accept an operation because an execution unit is busy.

For example a complex operation unit such as a divider is executing a previous operation and cannot accept a new operation.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when operations are available from the frontend but the backend is not able to accept an operation because an execution unit is busy for any PE in the multithreaded processor.

STALL_BACKEND_CPUBOUND, Backend stall cycles, processor bound

The counter counts each cycle counted by [STALL_BACKEND](#) when the backend is stalled on a processor resource, not including memory.

The counter counts each stall that occurs when a backend processor resource is busy. This includes the stalls counted by [STALL_BACKEND_RENAME](#), and any other IMPLEMENTATION DEFINED processor resource stalls.

It does not include stalls that are counted by [STALL_BACKEND_MEMBOUND](#), although both events might count on the same cycle counted by [STALL_BACKEND](#) if there are both memory and processor-resource stall conditions active.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when the backend is stalled on a processor resource, not including memory for any PE in the multithreaded processor.

STALL_BACKEND_GCSTLB, Backend stall cycles, GCS TLB

The counter counts each cycle counted by [STALL_BACKEND_MEMBOUND](#) when there is a demand data miss in a TLB.

The demand data miss must be due to translation of a GCS data read operation or GCS data write operation.

Implementation of this optional event requires implementing [LIGCS_TLB](#) event.

For a multithreaded processor implementation, if the Effective value of $\text{PMEVTYPER}_{<n>_EL0.MT}$ is 1, then the counter counts when there is a demand data miss in a TLB for any PE in the multithreaded processor.

STALL_BACKEND_ILOCK, Backend stall cycles, input dependency

The counter counts each cycle counted by [STALL_BACKEND](#) when operations are available from the frontend but at least one is not ready to be sent to the backend because of an input dependency.

For a multithreaded processor implementation, if the Effective value of $\text{PMEVTYPER}_{<n>_EL0.MT}$ is 1, then the counter counts when operations are available from the frontend but at least one is not ready to be sent to the backend because of an input dependency for any PE in the multithreaded processor.

STALL_BACKEND_L1D, Backend stall cycles, level 1 data cache

The counter counts each cycle counted by [STALL_BACKEND_MEMBOUND](#) when there is a demand data miss in the first level of data or unified cache.

For a multithreaded processor implementation, if the Effective value of $\text{PMEVTYPER}_{<n>_EL0.MT}$ is 1, then the counter counts when there is a demand data miss in the first level of data or unified cache for any PE in the multithreaded processor.

The counter does not count the cycle if any of the following are true:

- The [STALL_BACKEND_L2D](#) event is implemented and there is a demand data miss in the second level of data or unified cache, meaning the [STALL_BACKEND_L2D](#) event counts the cycle.
- There is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the [STALL_BACKEND_MEM](#) event counts the cycle.

STALL_BACKEND_L1GCS, Backend stall cycles, Level 1 GCS cache

The counter counts each cycle counted by [STALL_BACKEND_MEMBOUND](#) when there is a demand data miss in a first level cache.

The demand data miss must be due to either GCS data read operation or GCS data write operation.

The counter does not count the cycle if any of the following are true:

- The [STALL_BACKEND_L2D](#) event is implemented and there is a demand data miss in the second level of data or unified cache, meaning the [STALL_BACKEND_L2D](#) event counts the cycle.
- There is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the [STALL_BACKEND_MEM](#) event counts the cycle.

Implementation of this optional event requires that the first level cache is implemented within the PE clock domain and is not the last level cache within the PE clock domain.

Implementation of this optional event requires implementing [LIGCS_CACHE](#) event.

For a multithreaded processor implementation, if the Effective value of $\text{PMEVTYPER}_{<n>_EL0.MT}$ is 1, then the counter counts when there is a demand data miss in a first level cache for any PE in the multithreaded processor.

Note

It is possible that both the events [STALL_BACKEND_LIGCS](#) and [STALL_BACKEND_L1D](#) events may be counted on a same cycle.

STALL_BACKEND_L2D, Backend stall cycles, level 2 data cache

The counter counts each cycle counted by [STALL_BACKEND_MEMBOUND](#) when there is a demand data miss in the second level of data or unified cache.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when there is a demand data miss in the second level of data or unified cache for any PE in the multithreaded processor.

The counter does not count the cycle if there is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the [STALL_BACKEND_MEM](#) event counts the cycle.

STALL_BACKEND_MEM, Memory stall cycles

The counter counts each cycle counted by [STALL_BACKEND_MEMBOUND](#) when there is a demand data miss in the last level of data or unified cache within the PE clock domain.

It is IMPLEMENTATION DEFINED whether the counter counts backend stall cycles when a non-cacheable data access is in progress.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when there is a demand data miss in the last level of data or unified cache within the PE clock domain for any PE in the multithreaded processor.

`PMCEID0_EL0[37]` reads as 1 if this event is implemented and 0 otherwise.

STALL_BACKEND_MEMBOUND, Backend stall cycles, memory bound

The counter counts each cycle counted by [STALL_BACKEND](#) when the backend is waiting for a memory access to complete.

The counter counts each stall that occurs when the backend interface to memory is busy or stalled.

This includes the stalls counted by: [STALL_BACKEND_L1D](#), [STALL_BACKEND_L2D](#), [STALL_BACKEND_MEM](#), [STALL_BACKEND_ST](#), [STALL_BACKEND_TLB](#), and any other IMPLEMENTATION DEFINED memory stall events.

It does not include stalls that are counted by [STALL_BACKEND_CPUBOUND](#), although both events might count on the same cycle counted by [STALL_BACKEND](#) if there are both memory and processor-resource stall conditions active.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when the backend is waiting for a memory access to complete for any PE in the multithreaded processor.

STALL_BACKEND_RENAME, Backend stall cycles, rename full

The counter counts each cycle counted by [STALL_BACKEND_CPUBOUND](#) when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available.

If this event is implemented and counts such stalls then the [STALL_FRONTEND_RENAME](#) event counts as zero.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available for any PE in the multithreaded processor.

STALL_BACKEND_ST, Backend stall cycles, store

The counter counts each cycle counted by [STALL_BACKEND_MEMBOUND](#) when the backend is stalled waiting for a store.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when the backend is stalled waiting for a store for any PE in the multithreaded processor.

STALL_BACKEND_TLB, Backend stall cycles, TLB

The counter counts each cycle counted by [STALL_BACKEND_MEMBOUND](#) when there is a demand data miss in the data or unified TLB.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when there is a demand data miss in the data or unified TLB for any PE in the multithreaded processor.

STALL_FRONTEND_CPUBOUND, Frontend stall cycles, processor bound

The counter counts each cycle counted by [STALL_FRONTEND](#) when the frontend is stalled on a frontend processor resource, not including memory.

The counter counts each stall that occurs when a frontend processor resource is busy.

This includes the stalls counted by [STALL_FRONTEND_FLOW](#), [STALL_FRONTEND_FLUSH](#), and [STALL_FRONTEND_RENAME](#), and any other IMPLEMENTATION DEFINED processor resource stalls.

It does not include stalls that are counted by [STALL_FRONTEND_MEMBOUND](#). However both events will count the same cycle counted by [STALL_FRONTEND](#) if there are both memory and processor-resource stall conditions active.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when the frontend is stalled on a frontend processor resource, not including memory for any PE in the multithreaded processor.

STALL_FRONTEND_FLOW, Frontend stall cycles, flow control

The counter counts each cycle counted by [STALL_FRONTEND_CPUBOUND](#) when the frontend is stalled on unavailability of prediction flow resources.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when the frontend is stalled on unavailability of prediction flow resources for any PE in the multithreaded processor.

Note

This event is not counting stalls due to mispredictions, but rather stalls when the frontend is unable to make a prediction.

STALL_FRONTEND_FLUSH, Frontend stall cycles, flush recovery

The counter counts each cycle counted by [STALL_FRONTEND_CPUBOUND](#) when the frontend is recovering from a pipeline flush.

The situations where the frontend is flushed are IMPLEMENTATION DEFINED. For example, the frontend might be flushed on a branch misprediction or on a Context synchronization event.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when the frontend is recovering from a pipeline flush for any PE in the multithreaded processor.

STALL_FRONTEND_L1I, Frontend stall cycles, level 1 instruction cache

The counter counts each cycle counted by [STALL_FRONTEND_MEMBOUND](#) when there is a demand instruction miss in the first level of instruction cache.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when there is a demand instruction miss in the first level of instruction cache for any PE in the multithreaded processor.

The counter does not count the cycle if any of the following are true:

- The [STALL_FRONTEND_L2I](#) event is implemented and there is a demand instruction miss in the second level of instruction cache, meaning the [STALL_FRONTEND_L2I](#) event counts the cycle.
- There is a demand instruction miss in the last level of instruction cache within the PE clock domain, meaning the [STALL_FRONTEND_MEM](#) event counts the cycle.

STALL_FRONTEND_L2I, Frontend stall cycles, level 2 instruction cache

The counter counts each cycle counted by [STALL_FRONTEND_MEMBOUND](#) when there is a demand instruction miss in the second level of instruction or unified cache.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when there is a demand instruction miss in the second level of instruction or unified cache for any PE in the multithreaded processor.

The counter does not count the cycle if there is a demand instruction miss in the last level of instruction cache within the PE clock domain, meaning the [STALL_FRONTEND_MEM](#) event counts the cycle.

STALL_FRONTEND_MEM, Frontend stall cycles, last level PE cache or memory

The counter counts each cycle counted by [STALL_FRONTEND_MEMBOUND](#) when there is a demand instruction miss in the last level of instruction or unified cache within the PE clock domain or a non-cacheable instruction fetch in progress.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when there is a demand instruction miss in the last level of instruction or unified cache within the PE clock domain or a non-cacheable instruction fetch in progress for any PE in the multithreaded processor.

STALL_FRONTEND_MEMBOUND, Frontend stall cycles, memory bound

The counter counts each cycle counted by [STALL_FRONTEND](#) when no instructions are delivered from the memory system.

The counter counts each stall that occurs when the frontend interface to memory is busy or stalled.

This includes the stalls counted by [STALL_FRONTEND_L1I](#), [STALL_FRONTEND_L2I](#), [STALL_FRONTEND_MEM](#), and [STALL_FRONTEND_TLB](#), and any other IMPLEMENTATION DEFINED memory stalls.

It does not include stalls that are counted by [STALL_FRONTEND_CPUBOUND](#). However both events will count the same cycle counted by [STALL_FRONTEND](#) if there are both memory and processor-resource stall conditions active.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when no instructions are delivered from the memory system for any PE in the multithreaded processor.

STALL_FRONTEND_RENAME, Frontend stall cycles, rename full

The counter counts each cycle counted by [STALL_FRONTEND_CPUBOUND](#) when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available.

If this event is implemented and counts such stalls then the [STALL_BACKEND_RENAME](#) event counts as zero.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available for any PE in the multithreaded processor.

STALL_FRONTEND_TLB, Frontend stall cycles, TLB

The counter counts each cycle counted by [STALL_FRONTEND_MEMBOUND](#) when there is a demand instruction miss in the instruction or unified TLB.

For a multithreaded processor implementation, if the Effective value of `PMEVTYPER<n>_EL0.MT` is 1, then the counter counts when there is a demand instruction miss in the instruction or unified TLB for any PE in the multithreaded processor.

ST_ALIGN_LAT, Store with additional latency from alignment

The counter counts each [Memory-write operation](#) counted by [LDST_ALIGN_LAT](#).

`PMCEID1_EL0[34]` reads as 1 if this event is implemented and 0 otherwise.

ST_FIXED_BYTES_SPEC, Non-scalable store bytes speculatively executed

The counter counts each speculatively written byte counted by [LDST_FIXED_BYTES_SPEC](#).

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred. That is, the counter is incremented by:

- Half the value that the [LDST_FIXED_BYTES_SPEC](#) event counts if the operation is a swap, or compare and swap operation.
- The same as for [LDST_FIXED_BYTES_SPEC](#) if the operation is any other store operation, including an atomic store operation.

ST_FIXED_OPS_SPEC, Non-scalable store element operations speculatively executed

The counter counts each [Memory-write operation](#) counted by [LDST_FIXED_OPS_SPEC](#).

For each instruction, the counter is incremented by the number of operations specified by the instruction. That is, the counter is incremented by:

- Half the value that the [LDST_FIXED_OPS_SPEC](#) event counts if the operation is a store atomic, compare and swap, or swap operation.
- The same as for [LDST_FIXED_OPS_SPEC](#) if the operation is any other store operation.

SVE_FP_BF16_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point

The counter counts each SVE data-processing operation counted by [ASE_SVE_FP_BF16_MIN_SPEC](#) due to an instruction where the smallest type was BFloat16 floating-point.

SVE_FP_FP8_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is 8-bit floating-point

The counter counts each SVE data-processing operation counted by [ASE_SVE_FP_FP8_MIN_SPEC](#) due to an instruction where the smallest type was 8-bit floating-point.

SVE_FP_HP_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is half-precision floating-point

The counter counts each SVE data-processing operation counted by [ASE_SVE_FP_HP_MIN_SPEC](#) due to an instruction where the smallest type was half-precision floating-point.

SVE_FP_PREDUCE_SPEC, Floating-point operation speculatively executed, SVE pairwise add step or pairwise reduce step

The counter counts each [speculatively executed](#) floating-point pairwise add step or pairwise reduce step operation counted by [SVE_FP_VREDUCE_SPEC](#) due to any of the following instructions:

SVE_FP_SP_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is single-precision floating-point

The counter counts each SVE data-processing operation counted by [ASE_SVE_FP_SP_MIN_SPEC](#) due to an instruction where the smallest type was single-precision floating-point.

SVE_LUT_SPEC, Lookup table operation speculatively executed, SVE data processing

The counter counts each [speculatively executed](#) LUT operation counted by [SE_LUT_SPEC](#) that returns a value from a lookup table made up of a Z register which is indexed by values from a Z register.

That is, due to any of the following instructions:

SVE_SPEC, Operation speculatively executed, SVE data processing

The counter counts each operation counted by [ASE_SVE_SPEC](#) that is an SVE data-processing operation.

An SVE data-processing operation is any operation due to an instruction that operates on the SVE scalable vector and predicate registers that is not any of the following:

- A load or store operation. These are counted by [SVE_INST_SPEC](#).
- If the Cryptographic Extension is implemented, an operation counted by [CRYPTO_SPEC](#) as a Cryptographic data-processing operation.

If the Cryptographic Extension and FEAT_SVE2 are implemented, operations due to the SVE [PMULLB](#) and [PMULLT](#) (Q variants) instructions are not counted by [CRYPTO_SPEC](#) and are counted as SVE data-processing operations.

TRCEXTOUT0, Trace unit external output 0

The counter counts each event signaled by the trace unit on external event 0.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[48] reads as 1 if this event is implemented and 0 otherwise.

TRCEXTOUT1, Trace unit external output 1

The counter counts each event signaled by the trace unit on external event 1.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[49] reads as 1 if this event is implemented and 0 otherwise.

TRCEXTOUT2, Trace unit external output 2

The counter counts each event signaled by the trace unit on external event 2.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[50] reads as 1 if this event is implemented and 0 otherwise.

TRCEXTOUT3, Trace unit external output 3

The counter counts each event signaled by the trace unit on external event 3.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[51] reads as 1 if this event is implemented and 0 otherwise.

UOP_RETIRED, Micro-operation architecturally executed

The counter counts each micro-operation that would be executed in a Simple sequential execution of the program.

Unlike [OP_RETIRED](#), this event is not linked to the definition of [OP_SPEC](#), meaning it counts micro-operations that are created from other operations after those operations are counted by [OP_SPEC](#).

UOP_SPEC, Microarchitectural operation speculatively executed

The counter counts each [speculatively executed](#) microarchitectural operation.

ZA_ACTIVE, CPU cycles, ZA enabled

The counter counts each cycle counted by [CPU_CYCLES](#) when PSTATE.ZA is 1.

D7.4 Cycle event counting

The `CPU_CYCLES` event and the cycle counter, `PMCCNTR`, count cycles. The duration of a cycle is subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions.

It is IMPLEMENTATION SPECIFIC whether `CPU_CYCLES` and `PMCCNTR` count when the PE is in WFI or WFE state, even if the clocks are not stopped.

In addition, events such as `STALL`, `STALL_FRONTEND` and `STALL_BACKEND` that are defined to only count cycles that are counted by the `CPU_CYCLES` event have the same limitation.

D7.4.1 Cycle event counting in multithreaded implementations

Multithreaded implementations can have various forms, some examples of these are:

- *Simultaneous Multithreading* (SMT), where every PE thread is active on every `Processor cycle`.
- *Fine-grained Multithreading* (FGMT), also known as a Barrel processor, where one PE thread is active on each `Processor cycle`, and this changes regularly.
- *Switch on Event Multithreading* (SoEMT), also known as *Coarse-grained Multithreading* (CGMT), where high latency events cause the processor to switch the active PE thread.

In the above examples, active means that the PE might execute the instructions. A PE can be active but not executing instructions when no instruction is available or because of limited execution resources.

It is IMPLEMENTATION SPECIFIC whether a thread is active when the thread is in WFE or WFI state. This applies for all forms of multithreaded implementation.

When the PMU implementation supports multithreading, the `Effective value` of `PMEVTYPER<n>.MT` bit is 0, and the thread is not in WFE or WFI state, the `CPU_CYCLES` event only counts `Processor cycles` when the thread is active. For the example multithreaded implementations, this means that, if the event counter is enabled, event counting is not prohibited, and the thread is not in WFE or WFI state:

- For an SMT implementation, the `CPU_CYCLES` event counts every `Processor cycle`.
- For a particular FGMT implementation, that alternates between two threads on each `Processor cycle`, the `CPU_CYCLES` event counts every other `Processor cycle`.
- For a particular SoEMT implementation, that is waiting for a long latency operation, the `CPU_CYCLES` event does not count `Processor cycles`, as the PE thread is not active.

It is IMPLEMENTATION SPECIFIC whether `CPU_CYCLES` event counts `Processor cycles` when the `Effective value` of `PMEVTYPER<n>.MT` bit is 0, the thread is active, and the thread is in WFE or WFI state.

If the `Effective value` of `PMEVTYPER<n>.MT` bit is 1 and at least one thread is not in WFE or WFI state, then the `CPU_CYCLES` event counts each `Processor cycle`, and can only count a maximum of one for each `Processor cycle`.

If the `Effective value` of `PMEVTYPER<n>.MT` bit is 1, then all of the following apply for events that count PE cycles when a condition applies:

- The event does not count `Processor cycles` when all threads are in WFE or WFI state.
- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts `Processor cycles` when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts `Processor cycle` when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the `STALL`, `STALL_FRONTEND`, or `STALL_BACKEND` event. The second condition is any condition in addition to this. For example, for the `STALL_FRONTEND_L1I` event, the stall condition is `STALL_FRONTEND`, and the second condition is when there is a demand instruction miss in the first level of instruction cache. For the `STALL`, `STALL_FRONTEND`, and `STALL_BACKEND` events themselves, the second condition is the null TRUE condition.

Note

The cycle counter, [PMCCNTR](#), is not affected by whether the thread is active or inactive. When enabled, [PMCCNTR](#) counts every processor cycle.

D7.5 Meaningful combinations of common events

The meaningful combinations listed in this section comprise a numerator event and a denominator event.

In many cases, the definition of the numerator event, <EVENT_NUMERATOR>, is defined such that it counts each event <X> counted by the denominator event, <EVENT_DENOMINATOR>, when some condition <COND> is true.

This means that there is a meaningful ratio between between <EVENT_NUMERATOR> and <EVENT_DENOMINATOR> that describes the proportion of <X> counted by <EVENT_DENOMINATOR> for which <COND> applies.

This does not mean that there is a requirement that if <EVENT_NUMERATOR> is implemented that <EVENT_DENOMINATOR> is also implemented.

A *meaningful combination* means that any microarchitectural conditions that affect the counting of events apply equally to all the events. This typically means that they are counted at the same place in the pipeline, so that, for example, they are equally affected by pipeline flushes to discard speculatively executed operations following a mispredicted branch.

The following tables list the highlighted combinations:

- [Table D7-4](#), REFILL events and associated access events.
- [Table D7-5](#), Meaningful combinations of latency events.
- [Table D7-6](#), Cache hit events and associated data source events.
- [Table D7-7](#), Meaningful combinations of cache line state tracking events.
- [Table D7-8](#), Meaningful combinations of at-retirement events.

D7.5.1 Refill events and associated access events

IGSMLS

The following are meaningful combinations of refill events and associated access events:

Table D7-4 Meaningful combinations of refill events and associated access events

Numerator	Denominator	Ratio
0x0001 L1I_CACHE_REFILL	0x0014 L1I_CACHE	Attributable Level 1 instruction cache refill rate
0x0002 L1I_TLB_REFILL	0x0026 L1I_TLB	Attributable Level 1 instruction TLB refill rate
0x0003 L1D_CACHE_REFILL	0x0004 L1D_CACHE	Attributable Level 1 data or unified cache refill rate
0x0005 L1D_TLB_REFILL	0x0025 L1D_TLB	Attributable Level 1 data or unified TLB refill rate
0x0017 L2D_CACHE_REFILL	0x0016 L2D_CACHE	Attributable Level 2 data or unified cache refill rate
0x0028 L2I_CACHE_REFILL	0x0027 L2I_CACHE	Attributable Level 2 instruction cache refill rate
0x002A L3D_CACHE_REFILL	0x002B L3D_CACHE	Attributable Level 3 data or unified cache refill rate
0x002D L2D_TLB_REFILL	0x002F L2D_TLB	Attributable Level 2 data or unified TLB refill rate
0x002E L2I_TLB_REFILL	0x0030 L2I_TLB	Attributable Level 2 instruction TLB refill rate
0x0019 BUS_ACCESS	0x001D BUS_CYCLES	Attributable Bus accesses per cycle
0x0033 LL_CACHE_MISS	0x0032 LL_CACHE	Attributable Last Level data or unified cache refill rate
0x0034 DTLB_WALK	0x0025 L1D_TLB	Attributable data TLB miss rate
0x0035 ITLB_WALK	0x0026 L1I_TLB	Attributable instruction TLB miss rate
0x0037 LL_CACHE_MISS_RD	0x0036 LL_CACHE_RD	Attributable memory read operation miss rate
0x0038 REMOTE_ACCESS_RD	0x0031 REMOTE_ACCESS	Attributable read accesses to another socket in a multi-socket system

Numerator	Denominator	Ratio
0x0042 L1D_CACHE_REFILL_RD	0x0040 L1D_CACHE_RD	Attributable Level 1 cache refill rate, read
0x0043 L1D_CACHE_REFILL_WR	0x0041 L1D_CACHE_WR	Attributable Level 1 cache refill rate, write
0x004C L1D_TLB_REFILL_RD	0x004E L1D_TLB_RD	Attributable Level 1 TLB refill rate, read
0x004D L1D_TLB_REFILL_WR	0x004F L1D_TLB_WR	Attributable Level 1 TLB refill rate, write
0x0052 L2D_CACHE_REFILL_RD	0x0050 L2D_CACHE_RD	Attributable Level 2 data cache refill rate, read
0x0053 L2D_CACHE_REFILL_WR	0x0051 L2D_CACHE_WR	Attributable Level 2 data cache refill rate, write
0x005C L2D_TLB_REFILL_RD	0x005E L2D_TLB_RD	Attributable Level 2 data TLB refill rate, read
0x005D L2D_TLB_REFILL_WR	0x005F L2D_TLB_WR	Attributable Level 2 data TLB refill rate, write
0x00A2 L3D_CACHE_REFILL_RD	0x00A0 L3D_CACHE_RD	Attributable Level 3 data cache refill rate, read
0x00A3 L3D_CACHE_REFILL_WR	0x00A1 L3D_CACHE_WR	Attributable Level 3 data cache refill rate, write

D7.5.2 Latency events

I_{ZHFD}C The following are meaningful combination of latency events:

Table D7-5 Meaningful combinations of latency events

Numerator	Denominator	Ratio
0x8120 INST_FETCH_PERCYC	0x8124 INST_FETCH	Mean duration of instruction fetch events in processor cycles
0x8121 MEM_ACCESS_RD_PERCYC	0x0066 MEM_ACCESS_RD	Mean duration of memory read access events in processor cycles
0x8125, BUS_REQ_RD_PERCYC	0x818D, BUS_REQ_RD	Bus read transaction average latency
0x8128 DTLB_WALK_PERCYC	0x0034 DTLB_WALK	Mean duration of data or unified TLB walk events in processor cycles
0x8129 ITLB_WALK_PERCYC	0x0035 ITLB_WALK	Mean duration of instruction TLB walk events in processor cycles
0x8320, L1D_CACHE_REFILL_PERCYC	0x0003, L1D_CACHE_REFILL	Level 1 data cache refill average duration

D7.5.3 Cache hit events and associated data source events

I_{HMZQ}Y The following are meaningful combination of cache hit events:

Table D7-6 Meaningful combinations of cache hit events and associated data source events

Numerator	Denominator	Ratio
0x820C, L1D_CACHE_HIT_PRFM ^a	0x8142, L1D_CACHE_PRFM ^a	Level 1 data or unified cache software prefetch hit ratio ^a
0x81CC, L1D_CACHE_HIT_RW ^a	0x82A0, MEM_ACCESS_RW	Level 1 data or unified cache demand cache hit rate ^a
0x81CF, LL_CACHE_HIT_RW	0x82A0, MEM_ACCESS_RW	Last level cache demand cache hit rate

Numerator	Denominator	Ratio
0x8232, LOCAL_MEM_RW	0x82A0, MEM_ACCESS_RW	Local memory demand access rate
0x823B, REMOTE_MEM_RW	0x82A0, MEM_ACCESS_RW	Remote memory demand access rate
0x821D, L2D_CACHE_HITM_RW ^a	0x81CD, L2D_CACHE_HIT_RW ^a	Level 2 data or unified cache demand cache hit rate ^a

a. Similar ratios can be defined for other caches.

D7.5.4 Cache line state tracking events

I_{LGBBR}

The following are meaningful combination of cache line stake tracking events:

Table D7-7 Meaningful combinations of cache line state tracking events

Ratio	Meaning
1 - (0x81EC, L1D_CACHE_HIT_RW_FHWPRF ÷ 0x8154, L1D_CACHE_HWPRF) ^a	Level 1 data or unified cache data fetched by hardware prefetcher not used ^a
1 - (0x81DC, L1D_CACHE_HIT_RW_FPRFM ÷ 0x8142, L1D_CACHE_PRFM) ^a	Level 1 data or unified cache data fetched by software prefetch not used ^a
0x825C, L1D_LFB_HIT_RW_FPRFM ÷ 0x8142, L1D_CACHE_PRFM ^a	Level 1 data or unified cache data fetched late by software prefetch ^a

a. Similar ratios can be defined for other caches.

D7.5.5 At-retirement events

I_{CYQZJ}

The following are meaningful combinations of at-retirement events:

Table D7-8 Meaningful combinations of at-retirement events

Numerator	Denominator	Ratio
0x82B0, L1I_CACHE_HIT_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction hit Level 1 instruction cache
0x82E0, L2I_CACHE_MISS_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction missed Level 2 instruction cache
0x82EC, L1D_CACHE_HIT_LDST_RETIRED	0x0008, INST_RETIRED	Architecturally executed load or store instruction hit in Level 1 data cache
0x82C9, L2D_CACHE_HITM_ST_RETIRED	0x0008, INST_RETIRED	Architecturally executed store instruction hit modified data in Level 2 data cache
0x82B7, LL_CACHE_MISS_LD_RETIRED	0x0008, INST_RETIRED	Architecturally executed load instruction missed in Last level cache
0x82F0, ITLB_HIT_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction which hit in the ITLB
0x82F7, DTLB_WALK_LDST_RETIRED	0x82AA, LDST_ANY_RETIRED	Architecturally executed LDST instructions DTLB miss fraction
0x82AF, CRYPTO_RETIRED	0x0008, INST_RETIRED	Architecturally executed cryptographic data processing fraction

D7.5.6 Speculatively executed operation events

I_{RDHPB} The number of speculatively executed operations performed on individual scalar values can be determined from a pair of event counters. Combined multiply-add and multiply-subtract instructions are counted as two operations per element.

Table D7-9 Meaningful combinations of speculatively executed operation events

Ratio	Meaning
$0 \times 80C0, FP_SCALE_OPS_SPEC \times VL \div 128 + 0 \times 80C1, FP_FIXED_OPS_SPEC$	Total number of individual floating-point operations (any precision) performed
$0 \times 80C2, FP_HP_SCALE_OPS_SPEC \times VL \div 128 + 0 \times 80C3, FP_HP_FIXED_OPS_SPEC$	Total number of individual half-precision floating-point operations performed
$0 \times 80C4, FP_SP_SCALE_OPS_SPEC \times VL \div 128 + 0 \times 80C5, FP_SP_FIXED_OPS_SPEC$	Total number of individual single-precision floating-point operations performed
$0 \times 80C6, FP_DP_SCALE_OPS_SPEC \times VL \div 128 + 0 \times 80C7, FP_DP_FIXED_OPS_SPEC$	Total number of individual double-precision floating-point operations performed
$0 \times 80C8, INT_SCALE_OPS_SPEC \times VL \div 128 + 0 \times 80C9, INT_FIXED_OPS_SPEC$	Total number of individual integer operations performed
$0 \times 80CA, LDST_SCALE_OPS_SPEC \times VL \div 128 + 0 \times 80CB, LDST_FIXED_OPS_SPEC$	Total number of individual load/store accesses performed
$0 \times 80CC, LD_SCALE_OPS_SPEC \times VL \div 128 + 0 \times 80CD, LD_FIXED_OPS_SPEC$	Total number of individual load accesses performed
$0 \times 80CE, ST_SCALE_OPS_SPEC \times VL \div 128 + 0 \times 80CF, ST_FIXED_OPS_SPEC$	Total number of individual store accesses performed

D7.5.7 Bytes loaded and stored events

I_{NYJRH} The number of bytes speculatively loaded from memory or stored to memory can be determined from a pair of event counters.

Table D7-10 Meaningful combinations of speculative load or store events

Ratio	Meaning
$0 \times 80DA, LDST_SCALE_BYTES_SPEC \times VL \div 128 + 0 \times 80DB, LDST_FIXED_BYTES_SPEC$	Total number of bytes loaded from memory and stored to memory
$0 \times 80DC, LD_SCALE_BYTES_SPEC \times VL \div 128 + 0 \times 80DD, LD_FIXED_BYTES_SPEC$	Total number of bytes loaded from memory
$0 \times 80DE, ST_SCALE_BYTES_SPEC \times VL \div 128 + 0 \times 80DF, ST_FIXED_BYTES_SPEC$	Total number of bytes stored to memory

D7.6 Required events

When at least one event counter is implemented, the following common events are implemented:

- 0x0000, [SW_INCR](#), Instruction architecturally executed, Condition code check pass, software increment.
- 0x0011, [CPU_CYCLES](#), Cycle.
- When the PE includes a Level 1 data or unified cache:
 - 0x0003, [L1D_CACHE_REFILL](#), Level 1 data cache refill.
 - 0x0004, [L1D_CACHE](#), Level 1 data cache access.
- If the implementation includes program-flow prediction:
 - At least one of the following is implemented:
 - 0x0012, [BR_PRED](#), Predictable branch [Speculatively executed](#).
 - 0x0021, [BR_RETIRED](#), Instruction architecturally executed, branch.
 - At least one of the following is implemented:
 - 0x0010, [BR_MIS_PRED](#), Mispredicted or not predicted branch [Speculatively executed](#).
 - 0x0022, [BR_MIS_PRED_RETIRED](#), Branch instruction architecturally executed, mispredicted.
 - At least one of the following applies:
 - Both [BR_PRED](#) and [BR_MIS_PRED](#) are implemented.
 - Both [BR_RETIRED](#) and [BR_MIS_PRED_RETIRED](#) are implemented.

Note

Arm strongly recommends that both [BR_RETIRED](#) and [BR_MIS_PRED_RETIRED](#) are implemented. If the implementation does not include program-flow prediction, Arm recommends that these events are implemented as described in [Common microarchitectural events](#).

- At least one of:
 - 0x0008, [INST_RETIRED](#), Instruction architecturally executed.
 - 0x001B, [INST_SPEC](#), Operation [Speculatively executed](#).

Note

Arm strongly recommends that event 0x0008 is implemented.

When any of the following common events are implemented, all three of them are implemented:

- 0x003D, [STALL_SLOT_BACKEND](#), No operation sent for execution on a [Slot](#) due to the backend.
- 0x003E, [STALL_SLOT_FRONTEND](#), No operation sent for execution on a [Slot](#) due to the frontend.
- 0x003F, [STALL_SLOT](#), No operation sent for execution on a [Slot](#).

Arm strongly recommends that the following events are implemented:

- 0x003A, [OP_RETIRED](#).
- 0x003B, [OP_SPEC](#).
- 0x003D, [STALL_SLOT_BACKEND](#).
- 0x003E, [STALL_SLOT_FRONTEND](#).
- 0x003F, [STALL_SLOT](#).

D7.7 IMPLEMENTATION DEFINED event numbers

Arm recommends that implementers establish a standardized numbering scheme for their IMPLEMENTATION DEFINED events, with common definitions, and common event numbers, applied to all of their implementations. In general, the recommended approach is for standardization across implementations with common features. However, Arm recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

The Arm architecture guarantees not to define any event prefixed with *IMP_* as part of the standard Arm architecture.

Arm strongly recommends that at least the following classes of event are identified in the IMPLEMENTATION DEFINED events:

- Separating each of the [STALL_FRONTEND](#) and [STALL_SLOT_FRONTEND](#) events to count holes in instruction availability.
- Separating each of the [STALL_BACKEND](#) and [STALL_SLOT_BACKEND](#) events, to count, for example, cumulative duration of stalls, unavailability of execution resources, or missed superscalar issue opportunities.
- Miss rates for additional levels of caches and TLBs.
- Any external events passed to the PE through an IMPLEMENTATION DEFINED mechanism.
- Cumulative duration of a PSTATE. {A, I, F} interrupt mask set to 1.
- Cumulative occupancy for resource queues, such as data access queues, and entry/exit counts, so that average latencies can be determined, separating out counts for key resources that might exist. An implementation might also provide registers in the IMPLEMENTATION DEFINED space to further extend such counts, for example by specifying a minimum latency for an event to be counted.
- Any other microarchitectural features that the implementer considers are valuable to count.

The range of possible IMPLEMENTATION DEFINED event numbers is described in [The PMU event number space and common events](#).

It is not required that implementations document the behavior of all IMPLEMENTATION DEFINED PMU events, including which PMU event numbers are used for these IMPLEMENTATION DEFINED events. However, Arm strongly recommends that implementations do include and document IMPLEMENTATION DEFINED PMU events. In particular events related to microarchitectural characteristics of the implementation not covered by Common events.

Chapter D8

The Generic Timer in AArch32 state

This chapter describes the implementation of the Arm Generic Timer as an extension to an Armv8 AArch32 implementation. It includes an overview of the AArch32 System register interface to an Arm Generic Timer.

It contains the following sections:

- [About the Generic Timer in AArch32 state](#)
- [The AArch32 view of the Generic Timer](#)

D8.1 About the Generic Timer in AArch32 state

Figure D8-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *ARM Generic Interrupt Controller Architecture Specification* defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

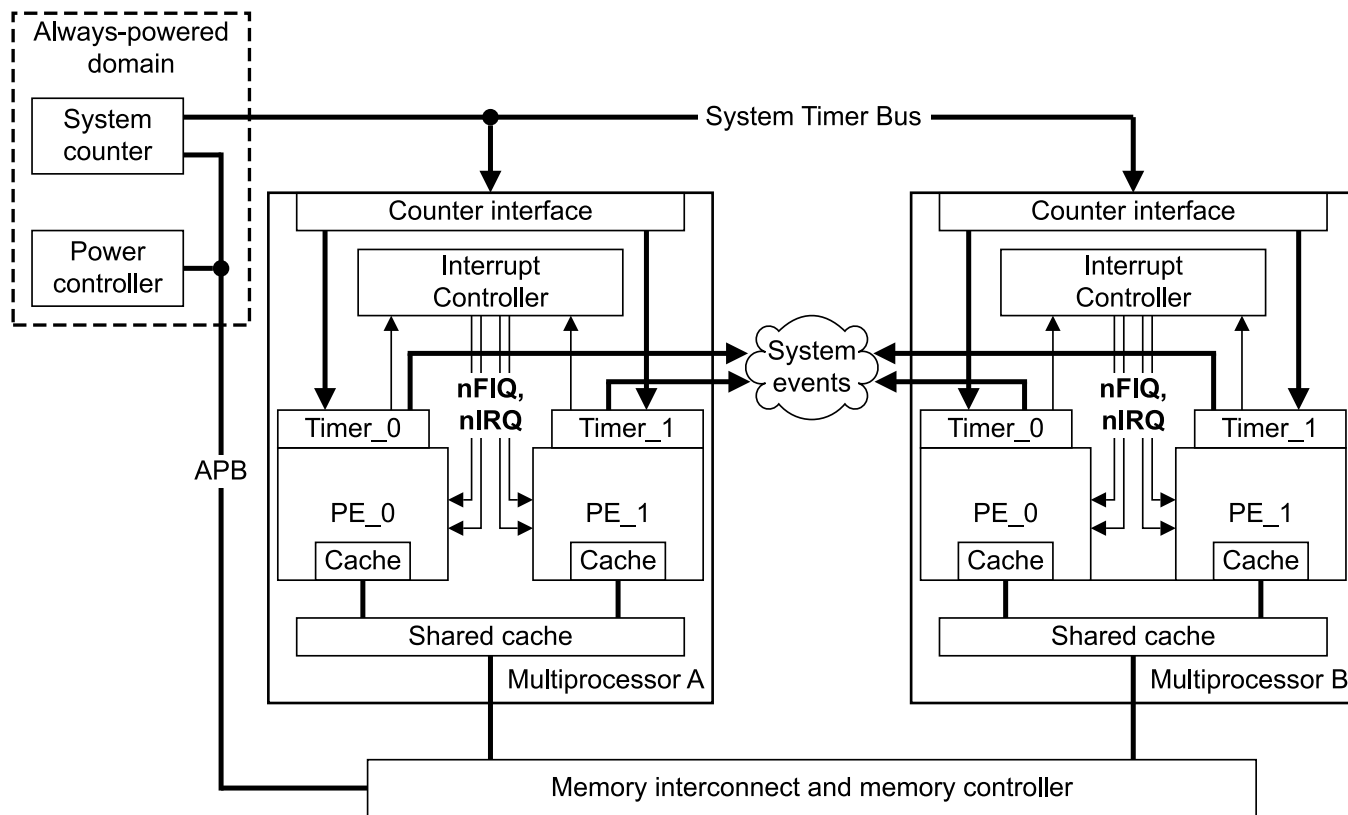


Figure D8-1 Generic Timer example

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.

Note

The Generic Timer can also provide other components at a system level, but Figure D8-1 does not show any such components.

- Timers, that can trigger events after a period of time has passed. The timers:
 - Can be used as count-up or as count-down timers.
 - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure D8-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block.

A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure D8-1.

- Optionally, can provide timer components for use at a system level.

[System Level Implementation of the Generic Timer](#) describes this memory-mapped component.

D8.1.1 The full set of Generic Timer components

Within a system that might include multiple PEs, a full set of Generic Timer components is as follows:

The system counter

This provides a uniform view of system time, see [The system counter](#). Because this must be implemented at the system level, it is accessed through [The system level memory-mapped implementation of the Generic Timer](#). However, during initialization, a status register in each implemented timer in the system must be programmed with the frequency of the system counter, so that software can read this frequency.

PE implementations of the Generic Timer

Each PE implementation of the Generic Timer provides the following components:

- A physical counter, that gives access to the count value of the system counter.
- A virtual counter, that gives access to virtual time. In AArch32 state, the [CNTVOFF](#) register defines the offset between physical time, as defined by the value of the system counter, and virtual time.
- A number of timers. In an implementation where all Exception levels are implemented and can use AArch32 state, the timers that are accessible from AArch32 state are:
 - An EL1 physical timer.
 - An EL2 physical timer.
 - An EL1 virtual timer.

[The AArch32 view of the Generic Timer](#) describes these components.

The system level memory-mapped implementation of the Generic Timer

The memory-mapped registers that control the components of the system level implementation of the Generic Timer are grouped into *frames*. The Generic Timer architecture defines the offset of each register within its frame, but the base address of each frame is IMPLEMENTATION DEFINED, and defined by the system.

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBaseN frame.

- Optionally, provides an unprivileged view of the timer in the CNTEL0Base N frame.

N is the timer number, and the corresponding frame number, in the range 0 - 7.

[System Level Implementation of the Generic Timer](#) describes these components.

D8.1.2 The system counter

The Generic Timer provides a system counter with the following specification:

Width

At least 56 bits wide. The value returned by any 64-bit read of the counter is zero-extended to 64 bits..

Effective frequency

This indicates the amount of time each unit of the counter represents. For example:

- An effective frequency of 50MHz means each unit of the counter represents 20ns.
- An effective frequency of 1GHz means each unit of the counter represents 1ns.
- The effective frequency is indicated in the [CNTFRQ](#) register.

The effective frequency is a fixed value, typically in the range 1-50MHz.

The counter might not be driven by a clock running at the effective frequency. For example, a counter with an effective frequency of 1GHz might have an actual clock running at 125MHz but the counter increments by 8 on each tick of the clock.

Roll-over

Roll-over time of not less than 40 years.

Counter resolution

The counter resolution is a representation of how frequently the counter is updated.

For example, a counter might have an [effective frequency](#) of 1GHz, but the actual clock runs at 125MHz and therefore the [counter resolution](#) is 125Mhz.

The counter resolution might vary over time, to accommodate different operating modes of the system such as lower power modes. Such variation might be invisible to the programmer and be system specific, or might be controllable by the programmer through the frequency modes table.

Accuracy

Arm does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.

Use of different counter resolutions must not affect the implemented accuracy.

Start-up

Starts operating from zero.

The system counter, once configured and running, must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes the counter can operate with a lower [counter resolution](#) by incrementing by larger amounts at a lower actual clock frequency. For example, a system counter with an [effective frequency](#) of 10MHz might either increment.

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the actual clock frequency, to reduce power consumption.

In this case, the counter must support transitions between higher resolution operation and lower resolution operation, without any impact on the required accuracy of the counter.

The [CNTFRQ](#) register is intended to hold a copy of the [effective frequency](#) to allow fast reference to this frequency by software running on the PE. For more information, see [Initializing and reading the system effective frequency](#).

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

Note

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. Gray count scheme for timer distribution scheme gives more information about this possibility.

D8.1.2.1 Initializing and reading the system effective frequency

The [CNTFRQ](#) register must be programmed to the [effective frequency](#) of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the system [effective frequency](#) to the [CNTFRQ](#) register. Only software executing at the highest implemented Exception level can write to [CNTFRQ](#).

Note

The [CNTFRQ](#) register is UNKNOWN at reset, and therefore the [effective frequency](#) must be set as part of the system boot process.

Software can read the [CNTFRQ](#) register, to determine the [effective frequency](#), in the following states and modes:

- Hyp mode.
- PL1 modes and EL1 modes.
- When [CNTKCTL](#).{PL0PCTEN, PL0VCTEN} is not {0,0}, EL0 modes.

D8.1.2.2 Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the [counter resolution](#).
- Enabling Halt-on-debug, which a debugger can then use to suspend counting.

For descriptions of these controls, see [System Level Implementation of the Generic Timer](#).

D8.2 The AArch32 view of the Generic Timer

The following sections describe the components and features of a PE implementation of the Generic Timer, as seen from AArch32 state:

- [The physical counter.](#)
- [The virtual counter.](#)
- [Event streams.](#)

D8.2.1 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The [CNTPCT](#) register holds the current physical counter value. For more information, see The physical offset register.

Reads of [CNTPCT](#) can occur speculatively and out of order relative to other instructions executed on the same PE.

D8.2.1.1 The self-synchronized view of the physical counter

Example D8-1 Ensuring reads of the physical counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that [CNTPCT](#) must be read, an ISB is used to ensure that the read of [CNTPCT](#) occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to
                   ; read the timer
    LDR R1, [R2]
    CMP R1, #1      ; has had the value 1 written to it
    BNE loop
    ISB             ; without this the CNTPCT could be read before the memory location in [R2]
    MRC R1, CNTPCT
```

Similarly where a read of the physical counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following code sequences can be used:

```
...                ; earlier loads and stores
    DSB             ; completes earlier loads and stores
    ISB             ; without this the CNTPCT could be read before the completion
                   ; of the earlier loads and stores
    MRC R1, CNTPCT
```

Neither view of the physical counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a [Dependency through registers and memory](#) from the read of the physical counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

Example D8-2 Ensuring reads of the physical counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

```
DSB
ISB
MRC Rn, CNTPCT          ; either view of the physical counter has the same effect in
                        ; this example
```

or

```
DSB
ISB
CBZ Ra, next
next
ISB                      ; this ISB is not needed if the MRC is accessing CNTPCT{SS}
MRC Rn, CNTPCT
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```
MRC Rn, CNTPCT          ; either view of the physical counter has the same effect
                        ; in this example
ISB
LDR Ra, [Rb]            ; this load will be executed after the timer has been read
```

or

```
MRC Rn, CNTPCT          ; either view of the physical counter has the same effect
                        ; in this example
EOR Rm, Rn, Rn
LDR Ra, [Rb]            ; this load will be executed after the counter has been read
```

D8.2.2 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, that indicates virtual time.

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing in a EL1 or EL0 mode, the virtual offset value relates to the current virtual machine.

The [CNTVOFF](#) register contains the virtual offset, see [The virtual offset register](#).

The [CNTVCT](#) register holds the current virtual counter value.

Reads of [CNTVCT](#) can occur speculatively and out of order relative to other instructions executed on the same PE.

D8.2.2.1 The self-synchronized view of the virtual counter

Example D8-3 Ensuring reads of virtual counter occur after signal read from memory

If a read from memory is used to obtain a signal from another agent that indicates that **CNTVCT** must be read, an **ISB** is used to ensure that the read of **CNTVCT** occurs after the signal has been read from memory, as shown in the following code sequence:

```
loop                ; polling for some communication to indicate a requirement to
                   ; read the timer

    LDR R1, [R2]
    CMP R1, #1      ; has had the value 1 written to it
    BNE loop
    ISB              ; without this the CNTVCT could be read before the memory
                   ; location in [R2]

    MRC R1, CNTVCT
```

Similarly where a read of the virtual counter is required to take place after the completion of all loads and stores appearing in program order before the read of the counter, then the following two sequences can be used:

```
...                ; earlier loads and stores
    DSB              ; completes earlier loads and stores
    ISB              ; without this the CNTVCT could be read before the completion
                   ; of the earlier loads and stores

    MRC R1, CNTVCT
```

Or, if **FEAT_ECV** is implemented:

```
...                ; earlier loads and stores
    DSB              ; completes earlier loads and stores

    MRC R1, CNTVCTSS
```

Neither view of the virtual counter ensures that:

- Context changes occurring in program order before the read of the counter have been synchronized.
- Accesses to memory appearing in program order after the read of the counter are executed before the counter has been read.

Where there is a [Dependency through registers and memory](#) from the read of the virtual counter to a Register effect generated by a read or write, the read or write will be executed after the read of the counter.

Example D8-4 Ensuring reads of virtual counter occur after previous memory accesses

To ensure that all previous memory accesses have completed and all previous context changes have been synchronized before the read of the counter, one of the following sequences should be used:

```
DSB
ISB
MRC Rn, CNTVCT      ; either view of the virtual counter has the same effect in
                    ; this example
```

or

```
DSB
ISB
CBZ Ra, next
next
ISB                  ; this ISB is not needed if the MRS is accessing CNTVCT{SS}
MRC Rn, CNTVCT
```

To ensure that a memory access occurs only after a read of the counter, then either of the following sequences should be used:

```
MRC Rn, CNTVCT      ; either view of the virtual counter has the same effect in
                    ; this example
ISB
LDR Ra, [Rb]         ; this load will be executed after the timer has been read
```

or

```
MRC Rn, CNTVCT      ; either view of the virtual counter has the same effect in
                    ; this example
EOR Rm, Rn, Rn
LDR Ra, [Rb]         ; this load will be executed after the counter has been read
```

D8.2.2.2 The virtual offset register

The virtual counter is a counter that has a *virtual offset* relative to the physical counter as viewed from EL2. This virtual offset is held in the register [CNTVOFF](#). The virtual counter value is the count compared by the EL1 virtual timer.

D8.2.3 Event streams

An implementation that includes the Generic Timer can use the system counter to generate one or more *event streams*, to generate periodic wakeup events as part of the mechanism described in Wait for Event.

Note

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.

The [CNTKCTL](#).{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the virtual counter.

In all implementations the [CNTHCTL](#).{EVNTEN, EVNTDIR, EVNTI, EVNTIS} fields define an event stream that is generated from the physical counter.

The event stream is configured as follows:

- EVNTI selects the counter bit that triggers the event.
- EVNTI selects between bits[0:15].

- EVNTDIR selects whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

Note

If the event stream is configured to produce events from the low order bits of the counter when the [effective frequency](#) is very high (for example 1GHz), then the practical update rate of the counter might mean that the event stream is not generated as the low order bit might not change. Software can rely on an event stream rate of at least 1MHz in normal operation.

The pseudocode descriptions of the operation of an event stream are [SetEventRegister](#), [TestEventCNTV](#), and [TestEventCNTP](#).

Chapter D9

AArch32 System Register Encoding

This chapter describes the AArch32 System register encoding space. It contains the following sections:

- [The AArch32 System register encoding space](#)
- [Organization of registers in the \(coproc == 0b1110\) encoding space](#)
- [Organization of registers in the \(coproc == 0b1111\) encoding space](#)

D9.1 The AArch32 System register encoding space

The T32 and A32 instruction sets includes instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- The cache, branch predictor, and TLB maintenance instructions, and address translation instructions.

[The AArch32 System register interface](#) describes the instructions that provide access to these registers and instructions. [AArch32 System Register Descriptions](#) describes these registers and encodings.

When accessing 32-bit registers, or executing these instructions, entries in the encoding space are characterized by the parameter set $\{\text{coproc}, \text{CRn}, \text{opc1}, \text{CRm}, \text{opc2}\}$. This encoding space is defined only for the `coproc` values 0b1110 and 0b1111.

Note

- When accessing 64-bit registers entries in the encoding space are characterized by the parameter set $\{\text{coproc}, \text{CRm}, \text{opc1}\}$, for the `coproc` values 0b1110 and 0b1111. A `CRm` value in this parameter set is equivalent to a `CRn` value in the parameter set for accessing 32-bit registers.
 - [Background to the System register interface](#) gives more information about this encoding model.
-

The following describe this encoding space:

- [Organization of registers in the \(`coproc == 0b1110` \) encoding space.](#)
- [Organization of registers in the \(`coproc == 0b1111` \) encoding space.](#)

D9.2 Organization of registers in the (coproc == 0b1110) encoding space

The System registers in the (coproc == 0b1110) encoding space provide a number of distinct control functions, covering:

- Debug.
- Trace.
- Execution environment control, for identification of the trivial Jazelle implementation.

Because these functions are distinct, the descriptions of these registers are distributed, as follows:

- In this manual, [Debug System registers](#) describes the Debug System registers.
- The *Embedded Trace Macrocell Architecture Specification* describes the Trace registers.

This section summarizes the allocation of the System registers in the (coproc == 0b1110) encoding space between these different functions, and the register encodings in this space that are reserved.

The 32-bit System register encodings are classified by the {opc1, CRn, opc2, CRm} values required to access them using an MCR or an MRC instruction. The 64-bit System register encodings are classified by the {opc1, CRm} values required to access them using an MCRR or an MRRC instruction. For the registers in the (coproc == 0b1110) encoding space, the opc1 value determines the primary allocation of these registers, as follows:

opc1==0 Debug System registers.

opc1==1 Trace registers.

opc1==7 Jazelle registers. Jazelle registers are implemented as required for a trivial Jazelle implementation.

Other opc1 values

Reserved.

Note

Primary allocation of (coproc == 0b1110) register function by opc1 value differs from the allocation of (coproc == 0b1111) registers, where primary allocation is by CRn value for 32-bit register accesses, or CRm value for 64-bit register accesses.

For the Debug System and Jazelle registers, [Table D9-1](#) defines:

- The {opc1, CRn, opc2, CRm} values used for accessing the 32-bit registers using the MRC and MCR instructions.
- The {opc1, CRm} values used for accessing the 64-bit register using the MRRC instruction.

Some Debug System registers can also be accessed using the LDC and STC instructions. [Table D9-1](#) defines the CRn values used for accessing the registers using these instructions.

Note

The only permitted uses of the LDC and STC instructions are:

- An LDC access to load data from memory to [DBGDTRTXint](#).
- An STC access to store data to memory from [DBGDTRRXint](#).

In the LDC and STC syntax descriptions in this Manual, the required coproc value of p14 and CRn value of c5 are given explicitly.

D9.2.1 Register access instruction arguments, (coproc == 0b1110) registers

Table D9-1 shows the MCR, MRC, and MRRC instruction arguments required for accesses to each register that can be visible in the System register interface in the (coproc == 0b1110) encoding.

Table D9-1 AArch32 PMSA (coproc == 0b1110) register summary, in MCR/MRC parameter order

coproc	opc1	CRn	CRm	opc2	Access	Mnemonic
0b1110	0b000	0b0000	0b0000	0b000	RO	DBGDIDR
0b1110	0b000	0b0000	0b0000	0b010	RW	DBGDTRRXext
0b1110	0b000	0b0000	0b0001	0b000	RO	DBGDSCRint
0b1110	0b000	0b0000	0b0010	0b000	RW	DBGDCCINT
0b1110	0b000	0b0000	0b0010	0b010	RW	DBGDSCRext
0b1110	0b000	0b0000	0b0011	0b010	RW	DBGDTRTXext
0b1110	0b000	0b0000	0b0101	0b000	RO	DBGDTRRXint
0b1110	0b000	0b0000	0b0110	0b000	RW	DBGWFAR
0b1110	0b000	0b0000	0b0110	0b010	RW	DBGOSECRR
0b1110	0b000	0b0000	0b0111	0b000	RW	DBGVCR
0b1110	0b000	0b0000	m[3:0]	0b100	RW	DBGBVR<m>
0b1110	0b000	0b0000	m[3:0]	0b101	RW	DBGBCR<m>
0b1110	0b000	0b0000	m[3:0]	0b110	RW	DBGWVR<m>
0b1110	0b000	0b0000	m[3:0]	0b111	RW	DBGWCR<m>
0b1110	0b000	0b0001	0b0000	0b000	RO	DBGDRAR
0b1110	0b000	0b0001	0b0001	0b100	RO	DBGOSLSR
0b1110	0b000	0b0001	0b0011	0b100	RW	DBGOSDLR
0b1110	0b000	0b0001	0b0100	0b100	RW	DBGPRCR
0b1110	0b000	0b0001	m[3:0]	0b001	RW	DBGBXVR<m>
0b1110	0b000	0b0010	0b0000	0b000	RO	DBGDSAR
0b1110	0b000	0b0111	0b0000	0b111	RO	DBGDEVID2
0b1110	0b000	0b0111	0b0001	0b111	RO	DBGDEVID1
0b1110	0b000	0b0111	0b0010	0b111	RO	DBGDEVID
0b1110	0b000	0b0111	0b1000	0b110	RW	DBGCLAIMSET
0b1110	0b000	0b0111	0b1001	0b110	RW	DBGCLAIMCLR
0b1110	0b000	0b0111	0b1110	0b110	RO	DBGAUTHSTATUS
0b1110	0b111	0b0000	0b0000	0b000	RO	JIDR
0b1110	0b111	0b0001	0b0000	0b000	RW	JOSCR
0b1110	0b111	0b0010	0b0000	0b000	RW	JMCR

Table D9-2 shows the LDC and STC instruction arguments required for accesses to the Debug registers that can be accessed using these instructions.

Table D9-2 Mapping of LDC and STC instruction arguments to System registers

Name	CRn	Instruction	Description
DBGDTRTXint	c5	LDC	Debug Data Transfer Register, Transmit, Internal View
DBGDTRRXint	c5	STC	Debug Data Transfer Register, Receive, Internal View

Note

In the instruction syntax descriptions for the LDC and STC instructions, the required *coproc* and *CRn* values are given explicitly as *coproc* == p14, *CRn* == c5.

D9.3 Organization of registers in the (coproc == 0b1111) encoding space

For 32-bit accesses to the System registers in the (coproc == 0b1111) encoding space, the ordered set of parameters {CRn, opc1, CRm, opc2} determine the register order. Within this ordering, the CRn value originally provided a functional grouping of these registers. As the number of System registers has increased this ordering has become less appropriate.

This document now:

- Groups the System registers in the (coproc == 0b1111) encoding space by functional group, see [Functional index of AArch32 registers and System instructions](#).

Note

Not all System registers introduced by architectural extensions are described in [AArch32 System Register Descriptions](#). For information about the System registers introduced by architectural extensions, see [R-profile Architecture Extensions](#).

This section presents information about the register ordering by {CRn, opc1, CRm, opc2}. It contains the following subsections:

- [System register summary for \(coproc == 0b1111\) encodings by CRn value.](#)
- [Full list of AArch32 PMSA System registers in the \(coproc == 0b1111\) encoding space.](#)

Note

The ordered listing of (coproc == 0b1111) registers by the {CRn, opc1, CRm, opc2} encoding of the 32-bit registers is most likely to be useful to those implementing AArch32 state, and to those validating such implementations. However, otherwise, the grouping of registers by function is more logical.

In addition, the indexes in [Registers Index](#) include all of the System registers.

D9.3.1 System register summary for (coproc == 0b1111) encodings by CRn value

[Figure D9-1](#) summarizes the grouping of the System registers in the (coproc == 0b1111) encoding space, for an AArch32 PMSA implementation, by the value of CRn used for a 32-bit access to the register.

CRn	opc1	CRm	opc2	
c0	{0-2, 4}	{c0-c2}	{0-7}	ID registers
c1	{0, 4}	{c0, c1}	{0-7}	System control registers
c2	{0, 4}	{c0, c1}	{0-2}	Memory system control registers
c3	0	c0	0	
c4	0	c6	0	GIC System register *, Debug exception registers
c5	{0, 4}	{c0,c1}	{0,1}	Memory system fault registers
c6	{0, 4}	c0	{0, 2, 4}	
c7	{0, 4}	Various	Various	Cache maintenance, address translations, legacy operations
c8	{0, 4}	Various	Various	TLB maintenance operations
c9	{0-7}	Various	{0-7}	Performance monitors
c10	{0-7}	Various	{0-7}	Memory mapping registers and TLB operations
c11	{0-7}	{c0-c8,c15}	{0-7}	Reserved for DMA operations for TCM access
c12	{0-2, 4, 6}	Various	{0,1}	System control registers, GIC System registers *
c13	{0, 4}	c0	{0-4}	Process, Context, and Thread ID registers
c14	{0-7}	{c0-c15}	{0-7}	Generic Timer registers *, Performance Monitors registers *
c15	{0-7}	{c0-c15}	{0-7}	IMPLEMENTATION DEFINED registers

* If implemented

Figure D9-1 AArch32 System register groupings for (coproc == 0b1111), for 32-bit registers

Note

For the System registers in the (coproc == 0b1111) encoding space, [Figure D9-1](#) gives only an overview of the assigned encodings for 32-bit registers for each of the CRn values c0 - c15. For more information, see:

- The full list of registers in the (coproc == 0b1111) encoding space, in [Full list of AArch32 System registers in the \(coproc == 0b1111 \) encoding space](#), for the definition of the assigned and unassigned encodings for that register.
 - The register definitions in [AArch32 System Register Descriptions](#) for any dependencies on the implemented Exception levels.
-

In general, System register accesses using an unallocated set of {CRn, opc1, CRm, opc2} values are UNDEFINED.

[Behavior of AArch32 System registers with \(coproc == 0b1111, CRn == c0 \)](#) described the only exceptions to this rule.

The 32-bit System registers with (coproc == 0b1111, CRn == c15), and the corresponding 64-bit System registers, are reserved for IMPLEMENTATION DEFINED registers. For more information, see [Reserved encodings in the AArch32 System register \(coproc == 0b1111 \) space](#).

D9.3.1.1 The HSTR.Tn trap on (coproc == 0b1111) System registers

When the value of [HSTR.Tn](#) is 1, Non-secure PL1 accesses to System registers in the (coproc == 0b1111) encoding space using a CRn or CRm value that corresponds to the value of Tn are trapped to EL2, even if the encoding is UNDEFINED when the value of [HSTR.Tn](#) is 0. This applies:

- For 32 bit register accesses when the value of Rn in the MCR or MRC instruction corresponds to Tn.
- For 64 bit register accesses when the value of Rm in the MCRR or MRRC instruction corresponds to Tn.

If there are matching System register encodings that are accessible from Non-secure EL0 then those accesses are also trapped to EL2 when the value of [HSTR.Tn](#) is 1.

D9.3.1.2 Behavior of AArch32 PMSA System registers with (coproc == 0b1111, CRn == c0)

In the (coproc == 0b1111) encoding space, the 32-bit System registers with (CRn == c0) provide device and feature identification.

[Table D9-3](#) shows all of the architecturally required System registers with { coproc == 0b1111, CRn == c0 }. The behavior of 32-bit System register encodings in this group that are not shown in the table, and encodings that are part of an unimplemented Exception level, depends on the value of opc1, and possibly on the value of CRm and opc2, as follows:

opc1 == 0 All write accesses to the encodings are UNDEFINED.

For read accesses:

- The following encodings return an UNKNOWN value:
 - CRm == 3, opc2 == {0, 1, 2}.
 - CRm == {4, 6, 7}, opc2 == {0, 1}.
 - CRm == 5, opc2 == {0, 1, 4, 5}.
- All other encodings are RES0.

opc1 > 0 All accesses to the encodings are UNDEFINED.

See also [Accesses to unallocated encodings in the \(coproc == 0b111x \) encoding space](#).

Note

Some of these registers were previously described as being part of the CPUID identification scheme, see [The CPUID identification scheme](#).

D9.3.1.3 Reserved encodings in the AArch32 System register (coproc == 0b1111) space

AArch32 state reserves a number of regions in the (coproc == 0b1111) encoding space for IMPLEMENTATION DEFINED System registers. These reservations are defined in terms of the encoding of 32-bit accesses to the System register encoding space. That is, they are defined by the reserved 32-bit {CRn, opc1, CRm, opc2} encodings.

Reserved encodings that do not have an IMPLEMENTATION DEFINED function are UNDEFINED.

The following subsections give more information about these reserved encodings:

- Reserved 32-bit encodings with {coproc == 0b1111, CRn == c9}.
- Reserved 32-bit encodings with {coproc == 0b1111, CRn == c10}.
- Reserved 32-bit encodings with {coproc == 0b1111, CRn == c11}.
- Reserved 32-bit encodings with {coproc == 0b1111, CRn == c15}.

D9.3.1.3.1 Reserved 32-bit encodings with {coproc == 0b1111, CRn == c9}

In the AArch32 encoding space, for 32-bit encodings with {coproc == 0b1111, CRn == c9}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc == 0b1111, CRn == c9, opc1 == {0 - 7}, opc2 == {0 - 7}, CRm == {c0 - c2, c5 - c8}} are reserved for IMPLEMENTATION DEFINED branch predictor, cache, and TCM operations.
- Encodings with {coproc == 0b1111, CRn == c9, opc1 == {0 - 7}, opc2 == {0 - 7}, CRm == c15} are reserved for IMPLEMENTATION DEFINED performance monitors.

Note

These are distinct from the OPTIONAL Arm Performance Monitors Extension, the registers for which use the encoding space {coproc == 0b1111, CRn == c9, opc1 == {0 - 7}, opc2 == {0 - 7}, CRm == {c12 - c14}}.

D9.3.1.3.2 Reserved 32-bit encodings with {coproc == 0b1111, CRn == c10}

In the AArch32 encoding space, for 32-bit encodings with {coproc == 0b1111, CRn == c10}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc == 0b1111, CRn == c10, opc == {0 - 7}, CRm == {c0, c1, c4, c8}} are reserved for IMPLEMENTATION DEFINED TLB lockdown operations.

D9.3.1.3.3 Reserved 32-bit encodings with {coproc == 0b1111, CRn == c11}

In the AArch32 encoding space, for 32-bit encodings with {coproc == 0b1111, CRn == c11}, the following encodings are reserved for IMPLEMENTATION DEFINED purposes:

- Encodings with {coproc == 0b1111, CRn == c11, opc == {0 - 7}, CRm == {c0 - c8, c15}} are reserved for IMPLEMENTATION DEFINED DMA operations for TCM access.

The remainder of the AArch32 {coproc == 0b1111, CRn == c11} encoding space is UNDEFINED.

D9.3.1.3.4 Reserved 32-bit encodings with {coproc == 0b1111, CRn == c15}

The AArch32 System register encodings are reserved with (coproc == 0b1111, CRn == c15) for IMPLEMENTATION DEFINED purposes, and there are no restrictions on the use of these encodings. The documentation of the Arm implementation must describe fully any registers implemented in the {coproc == 0b1111, CRn == c15} encoding space. Normally, for processor implementations by Arm, this information is included in the *Technical Reference Manual* for the processor.

Typically, an implementation uses the {coproc == 0b1111, CRn == c15} encodings to provide test features, and any required configuration options that are not covered by this Manual.

This reservation means that the AArch32 64-bit encodings with {coproc == 0b1111, CRm == c15} are also reserved for IMPLEMENTATION DEFINED purposes, without any restrictions on the use of these encodings.

D9.3.2 Full list of AArch32 System registers in the (coproc == 0b1111) encoding space

Table D9-3 shows the System registers in the (coproc == 0b1111) encoding space in the AArch32 PMSA, in the order of the {CRn, opc1, CRm, opc2} parameter values used in MCR or MRC accesses to the 32-bit registers:

- For MCR or MRC accesses to the 32-bit registers, CRn is the primary identifier of the target System register for the access. This applies, also, to MCR or MRC instructions that provide 32-bit accesses to a single word of a 64-bit System register.
- For MCRR or MRRC accesses to the 64-bit registers, CRm is the primary identifier of the target System register for the access. Table D9-3 orders the 64-bit registers with the 32-bit registers accessed using the same primary register identifier. For example, the 64-bit encoding of TTBR0, that is accessed with (CRm == c2), is listed with the 32-bit registers that are accessed with (CRn == c2).

Table D9-3 AArch32 PMSA (coproc == 0b1111) register summary, in MCR/MRC parameter order

coproc	opc1	CRn	CRm	opc2	Access	Mnemonic
0b1111	0b000	0b0000	0b0000	0b000	RO	MIDR
0b1111	0b000	0b0000	0b0000	0b001	RO	CTR
0b1111	0b000	0b0000	0b0000	0b010	RO	TCMTR
0b1111	0b000	0b0000	0b0000	0b011	RO	TLBTR
0b1111	0b000	0b0000	0b0000	0b100	RO	MPUIR
0b1111	0b000	0b0000	0b0000	0b101	RO	MPIDR
0b1111	0b000	0b0000	0b0000	0b110	RO	REVIDR
0b1111	0b000	0b0000	0b0001	0b000	RO	ID_PFR0
0b1111	0b000	0b0000	0b0001	0b001	RO	ID_PFR1
0b1111	0b000	0b0000	0b0001	0b010	RO	ID_DFR0
0b1111	0b000	0b0000	0b0001	0b011	RO	ID_AFR0
0b1111	0b000	0b0000	0b0001	0b100	RO	ID_MMFR0
0b1111	0b000	0b0000	0b0001	0b101	RO	ID_MMFR1
0b1111	0b000	0b0000	0b0001	0b110	RO	ID_MMFR2
0b1111	0b000	0b0000	0b0001	0b111	RO	ID_MMFR3
0b1111	0b000	0b0000	0b0010	0b000	RO	ID_ISAR0
0b1111	0b000	0b0000	0b0010	0b001	RO	ID_ISAR1
0b1111	0b000	0b0000	0b0010	0b010	RO	ID_ISAR2
0b1111	0b000	0b0000	0b0010	0b011	RO	ID_ISAR3
0b1111	0b000	0b0000	0b0010	0b100	RO	ID_ISAR4
0b1111	0b000	0b0000	0b0010	0b101	RO	ID_ISAR5
0b1111	0b000	0b0000	0b0010	0b110	RO	ID_MMFR4
0b1111	0b000	0b0001	0b0000	0b000	RW	SCTLR
0b1111	0b000	0b0001	0b0000	0b001	RW	ACTLR
0b1111	0b000	0b0001	0b0000	0b010	RW	CPACR
0b1111	0b000	0b0001	0b0000	0b011	RW	ACTLR2
0b1111	0b000	0b0001	0b0001	0b010	RW	NSACR
0b1111	0b000	0b0100	0b0110	0b000	RW	ICC_PMR

coproc	opc1	CRn	CRm	opc2	Access	Mnemonic
0b1111	0b000	0b0101	0b0000	0b000	RW	DFSR
0b1111	0b000	0b0101	0b0000	0b001	RW	IFSR
0b1111	0b000	0b0101	0b0001	0b000	RW	ADFSR
0b1111	0b000	0b0101	0b0001	0b001	RW	AIFSR
0b1111	0b000	0b0110	0b0000	0b000	RW	DFAR
0b1111	0b000	0b0110	0b0000	0b010	RW	IFAR
0b1111	0b000	0b0110	0b0010	0b001	RW	PRSEL
0b1111	0b000	0b0110	0b0011	0b000	RW	PRBAR
0b1111	0b000	0b0110	0b0011	0b001	RW	PRLAR
0b1111	0b000	0b0111	0b0100	0b000	RW	PAR
0b1111	0b000	0b1001	0b1100	0b000	RW	PMCR
0b1111	0b000	0b1001	0b1100	0b001	RW	PMCNTENSET
0b1111	0b000	0b1001	0b1100	0b010	RW	PMCNTENCLR
0b1111	0b000	0b1001	0b1100	0b011	RW	PMOVS
0b1111	0b000	0b1001	0b1100	0b101	RW	PMSEL
0b1111	0b000	0b1001	0b1100	0b110	RO	PMCEID0
0b1111	0b000	0b1001	0b1100	0b111	RO	PMCEID1
0b1111	0b000	0b1001	0b1101	0b000	RW	PMCCNTR
0b1111	0b000	0b1001	0b1101	0b001	RW	PMXEVTYPER
0b1111	0b000	0b1001	0b1101	0b010	RW	PMXVCNTR
0b1111	0b000	0b1001	0b1110	0b000	RW	PMUSERENR
0b1111	0b000	0b1001	0b1110	0b001	RW	PMINTENSET
0b1111	0b000	0b1001	0b1110	0b010	RW	PMINTENCLR
0b1111	0b000	0b1001	0b1110	0b011	RW	PMOVSSET
0b1111	0b000	0b1010	0b0010	0b000	RW	PRRR-MAIR0
0b1111	0b000	0b1010	0b0010	0b001	RW	NMRR-MAIR1
0b1111	0b000	0b1010	0b0011	0b000	RW	AMAIR0
0b1111	0b000	0b1010	0b0011	0b001	RW	AMAIR1
0b1111	0b000	0b1100	0b0000	0b000	RW	VBAR
0b1111	0b000	0b1100	0b0000	0b001	RO	RVBAR-MVBAR
0b1111	0b000	0b1100	0b0001	0b000	RO	ISR
0b1111	0b000	0b1100	0b1000	0b000	RO	ICC_IAR0
0b1111	0b000	0b1100	0b1000	0b010	RO	ICC_HPIR0
0b1111	0b000	0b1100	0b1000	0b011	RW	ICC_BPR0
0b1111	0b000	0b1100	0b1000	0b1:m[1:0]RW		ICC_AP0R<m>
0b1111	0b000	0b1100	0b1001	0b0:m[1:0]RW		ICC_AP1R<m>
0b1111	0b000	0b1100	0b1011	0b011	RO	ICC_RPR
0b1111	0b000	0b1100	0b1100	0b000	RO	ICC_IAR1
0b1111	0b000	0b1100	0b1100	0b010	RO	ICC_HPIR1
0b1111	0b000	0b1100	0b1100	0b011	RW	ICC_BPR1

coproc	opc1	CRn	CRm	opc2	Access	Mnemonic
0b1111	0b000	0b1100	0b1100	0b100	RW	ICC_CTLR
0b1111	0b000	0b1100	0b1100	0b101	RW	ICC_SRE
0b1111	0b000	0b1100	0b1100	0b110	RW	ICC_IGRPEN0
0b1111	0b000	0b1100	0b1100	0b111	RW	ICC_IGRPEN1
0b1111	0b000	0b1101	0b0000	0b000	RW	FCSEIDR
0b1111	0b000	0b1101	0b0000	0b001	RW	CONTEXTIDR
0b1111	0b000	0b1101	0b0000	0b010	RW	TPIDRURW
0b1111	0b000	0b1101	0b0000	0b011	RW	TPIDRURO
0b1111	0b000	0b1101	0b0000	0b100	RW	TPIDRPRW
0b1111	0b000	0b1110	0b0000	0b000	RW	CNTFRQ
0b1111	0b000	0b1110	0b0001	0b000	RW	CNTKCTL
0b1111	0b000	0b1110	0b0010	0b000	RW	CNTP_TVAL
0b1111	0b000	0b1110	0b0010	0b001	RW	CNTP_CTL
0b1111	0b000	0b1110	0b0011	0b000	RW	CNTV_TVAL
0b1111	0b000	0b1110	0b0011	0b001	RW	CNTV_CTL
0b1111	0b000	0b1110	0b10:m[4:3]	m[2:0]	RW	PMEVCNTR<m>
0b1111	0b000	0b1110	0b1111	0b111	RW	PMCCFILTR
0b1111	0b000	0b1110	0b11:m[4:3]	m[2:0]	RW	PMEVTYPER<m>
0b1111	0b001	0b0000	0b0000	0b000	RO	CCSIDR
0b1111	0b001	0b0000	0b0000	0b001	RO	CLIDR
0b1111	0b001	0b0000	0b0000	0b111	RO	AIDR
0b1111	0b00:n[4]	0b0110	0b1:n[3:1]	n[0]:0b00	RW	PRBAR<n>
0b1111	0b00:n[4]	0b0110	0b1:n[3:1]	n[0]:0b01	RW	PRLAR<n>
0b1111	0b010	0b0000	0b0000	0b000	RW	CSSELR
0b1111	0b011	0b0100	0b0101	0b000	RW	DSPSR
0b1111	0b011	0b0100	0b0101	0b001	RW	DLR
0b1111	0b100	0b0000	0b0000	0b000	RW	VPIDR
0b1111	0b100	0b0000	0b0000	0b100	RO	HMPUIR
0b1111	0b100	0b0000	0b0000	0b101	RW	VMPIDR
0b1111	0b100	0b0001	0b0000	0b000	RW	HSCTLR
0b1111	0b100	0b0001	0b0000	0b001	RW	HACTLR
0b1111	0b100	0b0001	0b0000	0b011	RW	HACTLR2
0b1111	0b100	0b0001	0b0001	0b000	RW	HCR
0b1111	0b100	0b0001	0b0001	0b001	RW	HDCR
0b1111	0b100	0b0001	0b0001	0b010	RW	HCPTR
0b1111	0b100	0b0001	0b0001	0b011	RW	HSTR
0b1111	0b100	0b0001	0b0001	0b100	RW	HCR2
0b1111	0b100	0b0001	0b0001	0b111	RW	HACR
0b1111	0b100	0b0010	0b0000	0b000	RW	VSCTLR
0b1111	0b100	0b0101	0b0001	0b000	RW	HADFSR

coproc	opc1	CRn	CRm	opc2	Access	Mnemonic
0b1111	0b100	0b0101	0b0001	0b001	RW	HAIFSR
0b1111	0b100	0b0101	0b0010	0b000	RW	HSR
0b1111	0b100	0b0110	0b0000	0b000	RW	HDFAR
0b1111	0b100	0b0110	0b0000	0b010	RW	HIFAR
0b1111	0b100	0b0110	0b0000	0b100	RW	HPFAR
0b1111	0b100	0b0110	0b0001	0b001	RW	HPRENr
0b1111	0b100	0b0110	0b0010	0b001	RW	HPRSELr
0b1111	0b100	0b0110	0b0011	0b000	RW	HPRBAR
0b1111	0b100	0b0110	0b0011	0b001	RW	HPRLAR
0b1111	0b100	0b1010	0b0010	0b000	RW	HMAIR0
0b1111	0b100	0b1010	0b0010	0b001	RW	HMAIR1
0b1111	0b100	0b1010	0b0011	0b000	RW	HAMAIRO
0b1111	0b100	0b1010	0b0011	0b001	RW	HAMAIR1
0b1111	0b100	0b1100	0b0000	0b000	RW	HVBAR
0b1111	0b100	0b1100	0b0000	0b010	RW	HRMR
0b1111	0b100	0b1100	0b1000	0b0:m[1:0]RW		ICH_AP0R<m>
0b1111	0b100	0b1100	0b1001	0b0:m[1:0]RW		ICH_AP1R<m>
0b1111	0b100	0b1100	0b1001	0b101	RW	ICC_HSRE
0b1111	0b100	0b1100	0b1011	0b000	RW	ICH_HCR
0b1111	0b100	0b1100	0b1011	0b001	RO	ICH_VTR
0b1111	0b100	0b1100	0b1011	0b010	RO	ICH_MISR
0b1111	0b100	0b1100	0b1011	0b011	RO	ICH_EISR
0b1111	0b100	0b1100	0b1011	0b101	RO	ICH_ELRSR
0b1111	0b100	0b1100	0b1011	0b111	RW	ICH_VMCR
0b1111	0b100	0b1100	0b110:m[3]	m[2:0]	RW	ICH_LR<m>
0b1111	0b100	0b1100	0b111:m[3]	m[2:0]	RW	ICH_LRC<m>
0b1111	0b100	0b1101	0b0000	0b010	RW	HTPIDR
0b1111	0b100	0b1110	0b0001	0b000	RW	CNTHCTL
0b1111	0b100	0b1110	0b0010	0b000	RW	CNTHP_TVAL
0b1111	0b100	0b1110	0b0010	0b001	RW	CNTHP_CTL
0b1111	0b10:n[4]	0b0110	0b1:n[3:1]	n[0]:0b00	RW	HPRBAR<n>
0b1111	0b10:n[4]	0b0110	0b1:n[3:1]	n[0]:0b01	RW	HPRLAR<n>

D9.3.2.1 About the GIC System registers

From version 3 of the GIC architecture specification, the specification defines three groups of System registers, identified by the prefix of the register name:

ICC_ GIC physical CPU interface System registers.

ICH_ GIC virtual interface control System registers.

ICV_ GIC Virtual CPU interface System registers.

Note

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

In AArch32, the GIC System registers are all in the (coproc == 0b1111) encoding space with (CRn == c12). The ICV_* registers have the same {CRn, opc1, CRm, op2} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

When implemented, the GIC System registers form part of an Arm processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

For more information, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3 and version 4* (ARM IHI 0069).

Chapter D10

AArch32 System Register Descriptions

This chapter describes each of the AArch32 System registers.

It contains the following sections:

- [About the AArch32 System registers](#)
- [General system control registers](#)
- [Debug System registers](#)
- [Performance Monitors registers](#)
- [Generic Timer registers](#)

D10.1 About the AArch32 System registers

For general information about the AArch32 System registers, see:

In AArch32 System Register Encoding:

- [Organization of registers in the \(coproc == 0b1110 \) encoding space.](#)
- [Organization of registers in the \(coproc == 0b1111 \) encoding space.](#)

In this chapter:

- [Fixed values in the System register descriptions.](#)
- [General behavior of System registers.](#)
- [Principles of the ID scheme for fields in ID registers.](#)
- [About AArch32 System register accesses.](#)

The remainder of this chapter describes the AArch32 System registers, in the following sections:

- [General system control registers.](#)
- [Debug System registers.](#)
- [Performance Monitors registers.](#)
- [Generic Timer registers.](#)

D10.1.1 Fixed values in the System register descriptions

See [Fixed values in AArch32 instruction and System register descriptions](#). This section defines how the glossary terms [RAZ](#), [RES0](#), [RAO](#), and [RES1](#) can be represented in the System register descriptions.

D10.1.2 General behavior of System registers

Except where indicated, System registers are 32-bits wide. As stated in [About the System registers for VMSAv8-32](#), there are some 64-bit registers, and these include cases where software can access either a 32-bit view or a 64-bit view of a register. The register summaries, and the individual register descriptions, identify the 64-bit registers and how they can be accessed.

The following sections give information about the general behavior of these registers:

- [Register names.](#)
- [Read-only bits in read/write registers.](#)
- [The CPUID identification scheme.](#)
- [IMPLEMENTATION DEFINED performance monitors.](#)
- [UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses.](#)
- [Read-only and write-only register encodings.](#)
- [Reset behavior of AArch32 System registers.](#)
- [Synchronization of changes to AArch32 System registers.](#)

Unless otherwise indicated, information in the listed sections applies to all AArch32 System registers

See also [About AArch32 System register accesses](#).

D10.1.2.1 Register names

The Arm architecture guarantees not to define any register name prefixed with *IMP_* as part of the standard Arm architecture.

Note

Arm strongly recommends that any register names created in the IMPLEMENTATION DEFINED register spaces be prefixed with *IMP_*, where appropriate.

D10.1.2.2 Read-only bits in read/write registers

Some read/write registers include bits that are read-only. These bits ignore writes.

D10.1.2.3 The CPUID identification scheme

The ID_* registers were originally called the CPUID identification scheme registers. However, functionally, there is no value in separating these registers from the slightly larger Identification registers functional group. See [Table H5-3](#) for a list of the ID_* registers.

D10.1.2.4 IMPLEMENTATION DEFINED performance monitors

The architecture reserves some additional System register encodings in the (coproc == 0b1111) encoding space for optional additional IMPLEMENTATION DEFINED performance monitors. [Table D10-1](#) shows the allocation of these encodings:

Table D10-1 Performance Monitors System register encoding allocations

CRn	opc1	CRm	opc2	Name	Type
c9	0-7	c12-c14	0-7	Performance Monitors Extension registers, see Performance monitors System registers	RW or RO ^a
		c15	0-7	IMPLEMENTATION DEFINED	^b

a. The table referenced in the [Name](#) entry shows the type of each of the OPTIONAL Performance Monitors Extension registers.

b. Access depends on the register or operation, and is IMPLEMENTATION DEFINED.

D10.1.2.5 UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for AArch32 System register accesses

This section defines UNPREDICTABLE and UNDEFINED behaviors for accesses to System registers, including those cases where the Armv8 behavior is CONSTRAINED UNPREDICTABLE.

In AArch32 state, the following operations are UNDEFINED:

- All LDC and STC accesses, except for the LDC access to [DBGDTRTXint](#) and the STC access to [DBGDTRRXint](#) specified in [Table D9-2](#).
- All MCRR and MRRC operations to the (coproc == 0b111x) encoding space, except for those explicitly defined as accessing 64-bit System registers specified in [Table D9-1](#) and [Table D9-3](#).

Unless otherwise indicated in the individual register descriptions:

- Reserved fields in registers are RES0.
- Assigning a reserved value to a field has a CONSTRAINED UNPREDICTABLE effect, see [Reserved values in System and memory-mapped registers and translation table entries](#).

The following subsections give more information about UNPREDICTABLE, CONSTRAINED UNPREDICTABLE, and UNDEFINED behavior for accesses to the (coproc == 0b111x) encoding space:

- [Accesses to unallocated encodings in the \(coproc == 0b111x \) encoding space.](#)
- [Additional rules for MCR and MRC accesses to System registers.](#)
- [Effects of EL2 on System register accesses.](#)

D10.1.2.5.1 Accesses to unallocated encodings in the (coproc == 0b111x) encoding space

In the Armv8 architecture, accesses to unallocated register encodings in the (coproc == 0b111x) encoding space are UNDEFINED.

Note

In Armv7, except for 32-bit registers encoded with a CRn value of c12, accesses to unallocated 32-bit registers were UNPREDICTABLE. The Armv8 CONSTRAINED UNPREDICTABLE behavior of these accesses is that they are UNDEFINED, see [Unallocated System register access instructions](#).

D10.1.2.5.2 Additional rules for MCR and MRC accesses to System registers

The following operations are CONSTRAINED UNPREDICTABLE for all encodings in the (coproc == 0b111x) encoding space:

- All MCR operations from the PC.
- All MRC operations to APSR_nzcv, except for the (coproc == 0b1110) MRC operation to APSR_nzcv from [DBGDSCRint](#).

The CONSTRAINED UNPREDICTABLE behavior of these operations is described in [Using R15 by instruction](#).

For registers and operations that are accessible from a particular Privilege level, any attempt to access those registers from a lower Privilege level is UNDEFINED.

Some individual registers can be made inaccessible by setting configuration bits, possibly including IMPLEMENTATION DEFINED configuration bits, to disable access to the register. The effects of the architecturally-defined configuration bits are defined individually in this manual. Unless explicitly stated otherwise in this manual, setting a configuration bit to disable access to a register results in the register becoming UNDEFINED for MRC and MCR accesses.

See also [Read-only and write-only register encodings](#).

D10.1.2.5.3 Effects of EL2 on System register accesses

EL2 introduces classes of System registers, described in Classification of System registers. Some of these classes of register are either:

- Accessible only from certain modes or states.
- Accessible from certain modes or states only when configuration settings permit the access.

Accesses to these registers that are not permitted are UNDEFINED, meaning execution of the register access instruction generates an Undefined Instruction exception.

Note

This section applies only to registers that are accessible from some modes and states. That is, it applies only to register access instructions using an encoding that, under some circumstances, would perform a valid register access.

The following register classes restrict access in this way:

EL2-mode System registers

This register class is defined only in an implementation that includes EL2.

EL2-mode registers are accessible only from Hyp mode.

All other accesses to these registers are UNDEFINED.

For more information, see Hyp mode read/write registers in the (coproc == 0b1111) encoding space and Hyp mode encodings for shared (coproc == 0b1111) System registers.

EL2-mode write-only operations

This register class is defined only in an implementation that includes EL2.

EL2-mode write-only operations are accessible only from Hyp mode.

Write accesses to these operations are UNDEFINED in modes other than Hyp mode.

For more information, see Hyp mode (coproc == 0b1111) write-only System instructions.

D10.1.2.6 Read-only and write-only register encodings

Some System registers are *read-only* (RO) or *write-only* (WO). For example:

- Most identification registers are read-only.
- Most encodings that perform an operation, such as a cache maintenance instruction, are write-only.

If a particular Privilege level defines a register to be:

- RO, then any attempt to write to that register, at that Privilege level, is UNDEFINED. This means that any access to that register with $L = 0$ is UNDEFINED.
- WO, then any attempt to read from that register, at that Privilege level, is UNDEFINED. This means that any access to that register with $L = 1$ is UNDEFINED.

For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED.

Note

This section applies only to registers that this manual defines as RO or WO. It does not apply to registers for which other access permissions are explicitly defined.

D10.1.2.7 Reset behavior of AArch32 System registers

Reset values apply only to RW registers and fields, however:

- Some RO registers or fields, including feature ID registers and some status registers or register fields, always return a known value.
- Some RW and RO registers or register fields return status information about the PE. Unless the register description indicates that the value is UNKNOWN on reset, a read of the register immediately after a reset returns valid information.
- Some RW and RO registers and fields are aliases of other registers or fields. In these cases, the reset behavior of the aliased register or field determines the value returned by a read of the register immediately after a reset.
- WO registers that only have an effect on writes do not have meaningful reset values. However, an access to a WO register might affect underlying state, and that state might have a defined reset value.
- IMPLEMENTATION DEFINED registers have IMPLEMENTATION DEFINED reset behavior.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception level that is using AArch32, see:

- [PE state on reset into AArch32 state](#).
- The appropriate Trace architecture specification, for the Trace System registers.

For a PE reset into an Exception level that is using AArch32, the architecture defines which AArch32 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and [PE state on reset into AArch32 state](#) summarizes these architectural requirements. Otherwise, RW registers reset to an architecturally unknown value.

D10.1.2.7.1 Pseudocode description of resetting System registers

The `AArch32_ResetControlRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and [PE state on reset into AArch32 state](#).

Note

For debug and trace System registers, this function resets registers as defined for the appropriate level of reset.

D10.1.2.8 Synchronization of changes to AArch32 System registers

In this section, *this PE* means the PE on which accesses are being synchronized.

Note

See [Definitions of direct and indirect reads and writes and their side-effects](#) for definitions of the terms *direct write*, *direct read*, *indirect write*, and *indirect read*.

A *direct write* to a System register might become visible at any point after the change to the register, but without a [Context Synchronization event](#) there is no guarantee that the change becomes visible.

Any direct write to a System register is guaranteed not to affect any instruction that appears, in program order, before the instruction that performed the direct write, and any direct write to a System register must be synchronized before any instruction that appears after the direct write, in program order, can rely on the effect of that write. The only exceptions to this are:

- All direct writes to the same register, using the same encoding, are guaranteed to occur in program order.
- All direct writes to a register are guaranteed to occur in program order relative to all direct reads of the same register using the same encoding.
- Any System register access that an Arm *Architecture Specification* or equivalent specification defines as not requiring synchronization.
- If an instruction that appears in program order before the direct write performs a memory access, such as a memory-mapped register access, that causes an indirect read or write to a register, that memory access is subject to the memory order model. In this case, if permitted by the memory order model, the instruction that appears in program order before the direct write can be affected by the direct write. For information about the memory order model, see [Definition of the memory model](#).

These rules mean that an instruction that writes to one of the address translation instructions described in Address translation instructions must be explicitly synchronized to guarantee that the result of the address translation instruction is visible in the [PAR](#).

Note

In this case, the direct write to the encoding of the address translation instruction causes an *indirect write* to the [PAR](#). Without a [Context Synchronization event](#) after the direct write, there is no guarantee that the indirect write to the [PAR](#) is visible.

Conceptually, the explicit synchronization occurs as the first step of any [Context Synchronization event](#). This means that if the operation uses the state that had been changed but not synchronized before the operation occurred, the operation is guaranteed to use the state as if it had been synchronized.

Note

- This explicit synchronization is applied as the first step of the execution of any instruction that causes the synchronization operation. This means it does not synchronize any effect of changes to the System registers that might affect the fetch and decode of the instructions that cause the operation, such as breakpoints or changes to translation tables.
 - For a synchronous exception, the control state in use at the time the exception is generated determines the exception syndrome information, and this syndrome information is not changed by this synchronization at the start of taking the exception.
-

Except for the register reads listed in [Registers with some architectural guarantee of ordering or observability](#), if no [Context Synchronization event](#) is performed, direct reads of System registers can occur in any order.

[Table D10-2](#) shows the synchronization requirement between two reads or writes that access the same System register. In the column headings, *First* and *Second* refer to:

- Program order, for any read or write caused by the execution of an instruction by this PE, other than a read or write caused by a memory access made by that instruction.
- The order of arrival of asynchronous reads or writes made by this PE relative to the execution of instructions by this PE.

In addition:

- For indirect reads or writes caused by an external agent, such as a debugger, the mechanism that determines the order of the reads or writes is defined by that external agent. The external agent can provide mechanisms that ensure that any read or write it makes arrives at the PE. These indirect reads and writes are asynchronous to software execution on the PE.
- For indirect reads or writes caused by memory-mapped reads or writes made by this PE, the ordering of the memory accesses is subject to the memory order model, including the effect of the memory type of the accessed memory address. This applies, for example, if this PE reads or writes one of its registers in a memory-mapped register interface.

The mechanism for ensuring completion of these memory accesses, including ensuring the arrival of the asynchronous read or write at the PE, is defined by the system.

Note

Such accesses are likely to be given a Device memory attribute, but requiring this is outside the scope of the architecture.

- For indirect reads or writes caused by autonomous asynchronous events that are counted, for example events caused by the passage of time, the events are ordered so that:
 - Counts progress monotonically.
 - The events arrive at the PE in finite time and without undue delay.

Table D10-2 Synchronization requirements for updates to System registers

First read or write	Second read or write	Context Synchronization event required
Direct read	Direct read	No
	Direct write	No
	Indirect read	No ^a
	Indirect write	No ^a , but see text in this section for exceptions
Direct write	Direct read	No
	Direct write	No
	Indirect read	Yes ^a
	Indirect write	No, but see text in this section for exceptions
Indirect read	Direct read	No
	Direct write	No
	Indirect read	No
	Indirect write	No
Indirect write	Direct read	Yes, but see text in this section for exceptions
	Direct write	No, but see text in this section for exceptions
	Indirect read	Yes, but see text in this section for exceptions
	Indirect write	No, but see text in this section for exceptions

a. Although no synchronization is required between a Direct write and a Direct read, or between a Direct read and an Indirect write, this does not imply that a Direct read causes synchronization of a previous Direct write. This means that the sequence Direct write followed by Direct read followed by Indirect read, with no intervening context synchronization, does not guarantee that the Indirect read observes the result of the Direct write.

If the indirect write is to a register that [Registers with some architectural guarantee of ordering or observability](#) shows as having some guarantee of the visibility of an indirect write, synchronization might not be required.

If a direct read or a direct write to a register is followed by an indirect write to that register that is caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the indirect write relative to the direct read or direct write.

If an indirect write caused by a direct write is followed by an indirect write caused by an external agent, or by an autonomous asynchronous event, or as a result of a memory-mapped write, then synchronization is required to guarantee the ordering of the two indirect writes.

Where an indirect write occurs as a side-effect of an access, this happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect. For other information about indirect writes after a direct read or a direct write, see [Definitions of direct and indirect reads and writes and their side-effects](#).

Note

Where a register has more than one encoding, a direct write to the register using a particular encoding is not an indirect write to the same register with a different encoding.

Where an indirect write is caused by the action of an external agent, such as a debugger, or by a memory-mapped read or write by the PE, then an indirect write by that agent to a register using a particular access mechanism, followed by an indirect read by that agent to the same register using the same access mechanism and address does not need synchronization.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

To guarantee the visibility of changes to some registers, additional operations might be required before the [Context Synchronization event](#). For such a register, the definition of the register identifies these additional requirements.

In this manual, unless the context indicates otherwise:

- *Accessing* a System register refers to a direct read or write of the register.
- *Using* a System register refers to an indirect read or write of the register.

D10.1.2.8.1 Registers with some architectural guarantee of ordering or observability

For the registers for which [Table D10-3](#) shows that the ordering of direct reads is guaranteed, multiple direct reads of a single register, using the same encoding, occur in program order without any explicit ordering.

For the registers for which [Table D10-3](#) shows that some observability of indirect writes is guaranteed, an indirect write to the register caused by an external agent, an autonomous asynchronous event, or as a result of a memory-mapped write, is both:

- Observable to direct reads of the register, in finite time, without explicit synchronization.
- Observable to subsequent indirect reads of the register without explicit synchronization.

These two sets of registers are similar, as [Table D10-3](#) shows:

Table D10-3 Registers with a guarantee of ordering or observability, VMSAv8-32

Register	Ordering of direct reads	Observability of indirect writes	Notes
ISR	Guaranteed	Guaranteed	Interrupt Status Register
DBGCLAIMCLR	Guaranteed	Guaranteed	Debug CLAIM registers
DBGCLAIMSET	-	Guaranteed	
DBGDTRRXint	Guaranteed	Guaranteed	Debug Communication Channel registers
DBGDTRTXint	-	Guaranteed	

Register	Ordering of direct reads	Observability of indirect writes	Notes
The DCC flags in DBGDSCRint	Guaranteed	Guaranteed	
CNTPCT	Guaranteed	Guaranteed	Generic Timer registers
CNTP_TVAL	Guaranteed	Guaranteed	
CNTVCT	Guaranteed	Guaranteed	
CNTV_TVAL	Guaranteed	Guaranteed	
CNTHP_TVAL	Guaranteed	Guaranteed	
PMCCNTR	Guaranteed	Guaranteed	Performance Monitors Extension registers, if the implementation includes the extension
PMEVCNTR<n>	Guaranteed	Guaranteed	
PMXEVCNTR	Guaranteed	Guaranteed	
PMOVSSET	Guaranteed	Guaranteed	
PMOVSr	Guaranteed	Guaranteed	
EDSCR .PipeAdv and the DCC flags in EDSCR	-	Guaranteed	Fields of the External Debug Status and Control Register

In addition to the requirements shown in [Table D10-3](#):

- Indirect writes to the following registers as a result of memory-mapped writes, including accesses by external agents, are required to be observable to the indirect read made in determining the response to a subsequent memory-mapped access without explicit synchronization:
 - [OSLAR_EL1](#). [OSLAR_EL1](#) is indirectly read to determine whether the subsequent access is permitted.

Note

[OSLAR_EL1](#) maps to the AArch32 System register [DBGOSLAR](#).
 - [EDLAR](#), if implemented. [EDLAR](#) is indirectly read to determine whether a subsequent write or side-effect of an access is ignored.

Note

This requirement is stricter than the general requirement for the observability of indirect writes.
- The requirement that an indirect write to the registers in [Table D10-3](#) is observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time. For example, an increment of the system counter is an autonomous asynchronous event that performs an indirect write to the counter. This asynchronous event might generate a timer interrupt request, resulting in a [Context Synchronization event](#). When a GIC is used, the timer interrupt might arrive at the GIC after the PE has taken an interrupt request from another source, but before software reads the current interrupt ID from the GIC. This means that the GIC might identify the timer interrupt as the current interrupt. Software must not assume that a subsequent direct read of the counter register is guaranteed to observe the updated value of that register. Although this example uses the counter-timer registers, it applies equally to other registers that might be linked to interrupt requests, including the PMU status registers.
- When the PE is in Debug state, there are synchronization requirements for the Debug Communication Channel and Instruction Transfer registers. See [DCC and ITR access in Debug state](#).

The possibility that direct reads can occur *early*, in the absence of context synchronization, described in [Ordering of reads of System registers](#), still applies to the registers listed in [Table D10-3](#).

D10.1.2.8.2 Definitions of direct and indirect reads and writes and their side-effects

Direct and indirect reads and writes are defined as follows:

Direct read Is a read of a register, using an MRC, MRRC, or STC instruction, that the architecture permits for the current PE state.

If a direct read of a register has a side-effect of changing the value of a register, the effect of a direct read on that register is defined to be an *indirect write*, and has the synchronization requirements of an indirect write. This means the indirect write is guaranteed to have occurred, and to be visible to subsequent direct or indirect reads and writes only if synchronization is performed after the direct read.

Note

The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.

Direct write

Is a write to a register, using an `MCR`, `MCCR`, or `LDC` instruction, that the architecture permits for the current PE state.

In the following cases, the side-effect of the direct write is defined to be an indirect write of the affected register, and has the synchronization requirements of an indirect write:

- If the direct write has a side-effect of changing the value of a register other than the register accessed by the direct write.
- If the direct write has a side-effect of changing the value of the register accessed by the direct write, so that the value in that register might not be the value that the direct write wrote to the register.

In both cases, this means that the indirect write is not guaranteed to be visible to subsequent direct or indirect reads and writes unless synchronization is performed after the direct write.

Note

- As an example of a direct write to a register having an effect that is an indirect write of that register, writing 1 to a `PMCNTENCLR.Px` bit is also an indirect write, because if the `Px` bit had the value 1 before the direct write, the side-effect of the write changes the value of that bit to 0.
 - The indirect write described here can affect either the register written to by the direct write, or some other register. The synchronization requirement is the same in both cases.

For example, writing 1 to a `PMCNTENCLR.Px` bit that is set to 1 also changes the corresponding `PMCNTENSET.Px` bit from 1 to 0. This means that the direct write to the `PMCNTENCLR` defines indirect writes to both itself and to the `PMCNTENSET`.
-

Indirect read

Is a use of the register by an instruction to establish the operating conditions for the instruction. Examples of operating conditions that might be determined by an indirect read are the translation table base address, or whether memory accesses are forced to be Non-cacheable.

Indirect reads include situations where the value of one register determines what value is returned by a second register. This means that any read of the second register is an indirect read of the register that determines what value is returned.

Indirect reads also include:

- Reads of the System registers by external agents, such as debuggers, as described in [Debug System registers](#).
- Memory-mapped reads of the System registers made by the PE on which the System registers are implemented.

Where an indirect read of a register has a side-effect of changing the value of a register, that change is defined to be an indirect write, and has the synchronization requirements of an indirect write.

Indirect write

Is an update to the value of a register as a consequence of either:

- An exception, operation, or execution of an instruction that is not a direct write to that register.
- The asynchronous operation of an external agent.

This can include:

- The passage of time, as seen in counters or timers, including performance counters.
- The assertion of an interrupt.
- A write from an external agent, such as a debugger.

However, for some registers, the architecture gives some guarantee of visibility without any explicit synchronization, see [Registers with some architectural guarantee of ordering or observability](#).

Note

Taking an exception is a [Context Synchronization event](#). Any indirect write performed as part of an exception entry does not require additional synchronization. This includes the indirect writes to the registers that report the exception, as described in [Exception reporting in a VMSAv8-32 implementation](#).

D10.1.3 Principles of the ID scheme for fields in ID registers

The Arm architecture specifies a number of *ID registers* that are characterized as comprising a set of 4-bit *ID fields*. Each ID field identifies the presence, and possibly the level of support for, a particular feature in an implementation of the architecture. These fields follow an architectural model that aids their use by software and provides future compatibility. This section describes that model. [AArch32 ID registers to which this scheme applies](#) identifies the set of ID registers that are accessible from AArch32 state.

A small number of ID fields do not follow the scheme described in this section. In these cases, the field description states that it does not follow this scheme.

Note

- The ID fields described here are unlike the ones that enumerate the number of resources, such as the number of breakpoints, watchpoints, or performance monitors.
 - ID fields that do not follow this scheme include the [ID_DFR0.PerfMon](#) and [EDDFR.PMUVer](#) fields, see [Alternative ID scheme used for the Performance Monitors Extension version](#).
 - The presence of an ID field for a feature does not imply that the feature is optional.
-

To provide forward compatibility, software can rely on the features of these fields that are described in this section.

The ID fields, which are either signed or unsigned, use increasing numerical values to indicate increases in functionality. Therefore, if a value of 0x1 indicates the presence of some instructions, then the value 0x2 will indicate the presence of those instructions plus some additional instructions or functionality. This means software can be written in the form:

```
if (value >= number) {  
    // do something that relies on the value of the feature  
}
```

For ID fields where the value 0x0 defines that a feature is not present, the field holds an unsigned value. This covers the vast majority of such fields.

In a few cases, the architecture has been changed to permit implementations to exclude a feature that has previously been required and for which no ID field has been defined. In these cases, a new ID field is defined and:

- The field holds a signed value.
- The field value 0xF indicates that the feature is not implemented.
- The field value 0x0 indicates that the feature is implemented.
- Software that depends on the feature can use the test:

```
if value >= 0 {  
    // Software features that depend on the presence of the hardware feature  
}
```

In some cases, it has been decided retrospectively that the increase in functionality between two consecutive numerical values is too great, and it is desirable to permit an intermediate degree of functionality, and the means to discover this. This is done by the introduction of a *fractional* field that both:

- Is referred to in the definition of the original field.
- Applies only when the original field is at the lower value of the step.

In principle, a fractional field can be used for two different fractional steps, with different meanings associated with each of these steps. For this reason, a fractional field must be interpreted in the context of the field to which it relates and the value of that field. [Example D10-1](#) shows the use of such a field.

Example D10-1 Example of the use of a fractional field

For a field describing some class of functionality:

- The value `0x1` was defined as indicating that item A is present.
- The value `0x2` was defined as indicating that items B and C are present, in addition to item A.

Subsequently, it might be necessary to introduce a second ID field to indicate that A and B only are present. This new field is a fractional field, and might be defined as having the value `0x1` when A and B only are present. This fractional field is valid only when the original ID field has the value `0x1`.

This approach means that:

- Software that depends on the test `if (value >= 0x2)` can rely on features A, B, and C being present,
- Software that depends on the test `if (value >= 0x1)` can rely on feature A being present.
- If new software needs to check only that features A and B are present, then it can test:

```
if (value >= 0x2 || (value == 0x1 && fractional_value >= 0x1)) {
    // Software features that depend on A and B only
}
```

A fractional field uses the same approach of increasing numerical values indicating increasing functionality, and the fractional approach can also be applied recursively to fractional fields.

Unused ID fields, and fractional fields that are not applicable, are RES0 to allow their future use when features, or fractional implementation options, are added.

D10.1.3.1 AArch32 ID registers to which this scheme applies

- The Auxiliary Feature register [ID_AFR0](#).
- The Processor Feature registers [ID_PFR0](#) and [ID_PFR1](#).
- The Debug Feature register [ID_DFR0](#).
- The Memory Model Feature registers [ID_MMFR0](#), [ID_MMFR1](#), [ID_MMFR2](#), [ID_MMFR3](#), and [ID_MMFR4](#).
- The Instruction Set Attribute registers [ID_ISAR0](#), [ID_ISAR1](#), [ID_ISAR2](#), [ID_ISAR3](#), [ID_ISAR4](#), and [ID_ISAR5](#).
- The Media and VFP Feature registers [MVFR0](#), [MVFR1](#), and [MVFR2](#).

Note

Principles of the ID scheme for fields in ID registers includes information about the AArch64 System registers and the memory-mapped registers to which this scheme applies.

D10.1.3.2 Alternative ID scheme used for the Performance Monitors Extension version

The [ID_DFR0_EL1.PerfMon](#), [ID_DFR0.PerfMon](#), and [EDDFR.PMUVer](#) fields, which identify the version of the Performance Monitors Extension, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value `0xF` indicates that the Arm-architected Performance Monitors Extension is not implemented.
- If the field value is not `0xF` the field is treated as an unsigned value, as described for the standard ID scheme.

This means that software that depends on the implementation of a particular version of the Arm Performance Monitors Extension must be written in the form:

```
if (value != 0xF and value >= number) {
    // do something that relies on version 'number' of the feature
}
```

For these fields, Arm deprecates use of the value 0xF in new implementations.

D10.1.4 About AArch32 System register accesses

The following subsections give more information about accesses to the AArch32 System registers:

- [Ordering of reads of System registers.](#)
- [Accessing 32-bit System registers.](#)
- [Accessing 64-bit System registers.](#)

D10.1.4.1 Ordering of reads of System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

- Any data dependencies between the instructions, as specified in [Synchronization of changes to AArch32 System registers](#), including read-after-read dependencies, are respected.
- The reads to the register do not occur earlier than the most recent [Context Synchronization event](#) to its architectural position in the instruction stream.

Note

In particular, the values read from System registers that hold self-incrementing counts, such as the Performance Monitors counters or the Generic Timer counter or timers, could be accessed from any time after the previous [Context Synchronization event](#). For example, where a memory access is used to communicate a read of such a counter, an ISB must be inserted between the read of the memory location that is known to have returned its data, either as a result of a condition on that data or of the read having completed, and the read of the counter, if it is necessary that the counter returns a count value after the memory communication.

D10.1.4.2 Accessing 32-bit System registers

Software accesses most 32-bit System registers using the generic MCR and MRC System register access instructions, specifying some or all of the parameters {coproc, CRn, opc1, CRm, opc2}, where:

coproc	Identifies the primary region of the System register encoding space. Takes one of the values:	
	p14	Encoded as 0b1110.
	p15	Encoded as 0b1111.
CRn	Takes a value in the range c0- c15, encoded the corresponding 4-bit binary value, 0b0000 - 0b1111.	
	In the (coproc== 0b1110) encoding space, the opc1 value identifies the System register functional group, and CRn is the most significant identifier for the required register within that group.	
	In the (coproc== 0b1111) encoding space, CRn is the most significant identifier for the required register.	
opc1	Takes a value in the range 0 - 7, encoded as its 3-bit binary value.	
	In the (coproc== 0b1110) encoding space, the opc1 value identifies the System register functional group, and can take the following values:	
	0	Debug System registers.
	1	Trace System registers.
	7	Legacy Jazelle System registers.

In the (`coproc == 0b1111`) encoding space, `opc1` can take any value in the range 0 - 7.

<code>CRm</code>	Takes a value in the range <code>c0- c15</code> , encoded the corresponding 4-bit binary value, <code>0b0000 - 0b1111</code> .
<code>opc2</code>	Takes a value in the range 0 - 7, encoded as its 3-bit binary value. <code>opc2</code> is optional in the <code>MCR</code> and <code>MRC</code> instruction syntax, and if no value is specified the encoding defaults to <code>0b000</code> .
<code>Rt</code>	A general-purpose register to hold a 32-bit value to transfer to or from the System register. Takes a value in the range <code>R0 - R14</code> , encoded as the corresponding 4-bit binary value, <code>0b0000 - 0b1110</code> .

This means an `MCR` or `MRC` access to a specific 32-bit System register uses:

- A unique combination of `coproc`, `CRn`, `opc1`, `CRm`, and `opc2`, to specify the required System register.
- A general-purpose register, `Rt`, for the transferred 32-bit value.

See also:

- [MCR](#).
- [MRC](#).

A small number of AArch32 debug System registers are accessed using `LDC` or `STC` instructions. In these cases, the register to be accessed is identified in the instruction syntax by the use of `p14`, `c5` where:

<code>p14</code>	Identifies that the access is to the (<code>coproc == 0b1110</code>) encoding space.
<code>c5</code>	Identifies the target debug System register.

See the instruction descriptions:

- [LDC \(immediate\)](#).
- [LDC \(literal\)](#).
- [STC](#).

The only uses of `LDC` and `STC` permitted in Armv8-A are:

- An `LDC` access to load data from memory to [DBGDTRTXint](#), see [LDC \(immediate\)](#) and [LDC \(literal\)](#).
- An `STC` access to store data to memory from [DBGDTRRXint](#), see [STC](#).

A small number of AArch32 System registers are accessed using `MRS`, `MSR`, `VMRS`, or `VMSR` instructions, see the appropriate register and instruction description for more information, see:

- [MRS](#).
- [MSR \(immediate\)](#),
- [MSR \(register\)](#).
- [VMRS](#).
- [VMSR](#).

Note

- For example:
 - The [APSR](#), [CPSR](#), and [SPSR](#) are accessed using `MRS` or `MSR` instructions.
 - The [MVFR0](#), [MVFR1](#), and [MVFR2](#) are accessed using `VMRS` or `VMSR` instructions.
 - In addition, the banked register forms of the `MRS` and `MSR` instructions can be used to access some System registers associated with PE modes other than the mode in which the PE is currently executing, see [MRS \(Banked register\)](#) and [MSR \(Banked register\)](#).
-

D10.1.4.3 Accessing 64-bit System registers

Software accesses a 64-bit System register using the generic MCRR and MRRC System register access instructions, specifying the parameters {coproc, CRm, opc1}, where:

coproc	Identifies the primary region of the System register encoding space. Takes one of the values:	
	p14	Encoded as 0b1110.
	p15	Encoded as 0b1111.
CRm	Takes a value in the range c0- c15, encoded the corresponding 4-bit binary value, 0b0000 - 0b1111.	
	In the (coproc== 0b1110) encoding space, the opc1 value identifies the System register functional group, and CRm is the most significant identifier for the required register within that group.	
	In the (coproc== 0b1111) encoding space, CRm is the most significant identifier for the required register.	
opc1	Takes a value in the range 0 - 15, encoded as its 3-bit binary value.	
	In the (coproc== 0b1110) encoding space, the opc1 value identifies the System register functional group, and can take the following values:	
	0	Debug System registers.
	1	Trace System registers.
	In the (coproc== 0b1111) encoding space, opc1 can take any value in the range 0 - 15.	
Rt	A general-purpose register to hold bits[31:0] of the value to transfer to or from the System register. Takes a value in the range R0 - R14, encoded as the corresponding 4-bit binary value, 0b0000 - 0b1110.	
Rt2	A general-purpose register to hold bits[63:32] of the value to transfer to or from the System register. Takes a value in the range R0 - R14, encoded as the corresponding 4-bit binary value, 0b0000 - 0b1110.	

This means an MCRR or MRRC access to a specific 64-bit System register uses:

- A unique combination of coproc, CRm and opc1, to specify the required 64-bit System register.
- Two general-purpose registers, each holding 32 bits of the value to transfer.

This means a PE can access a 64-bit System register using:

- An MCRR instruction to write to a System register, see [MCRR](#).
- An MRRC instruction to read a System register, see [MRRC](#).

When using an MCRR or MRRC instruction the System register access is 64-bit atomic.

Some 64-bit registers also have an MCR and MRC encoding. The MCR and MRC encodings for these registers access the least significant 32 bits of the register. For example, to access the [PAR](#), software can:

- Use the following instructions to access all 64 bits of the register:

```
MRRC p15, 0, <Rt>, <Rt2>, c7 ; Read 64-bit PAR into Rt (low word) and Rt2 (high word)
```

```
MCRR p15, 0, <Rt>, <Rt2>, c7 ; Write Rt (low word) and Rt2 (high word) to 64-bit PAR
```

- Use the following instructions to access the least-significant 32 bits of the register:

```
MRC p15, 0, <Rt>, c7, c4, 0 ; Read PAR[31:0] into Rt
```

```
MCR p15, 0, <Rt>, c7, c4, 0 ; Write Rt to PAR[31:0]
```

D10.2 General system control registers

This section lists the System registers in AArch32 state that are not part of one of the other listed groups.

D10.2.1 ACTLR, Auxiliary Control Register

The ACTLR characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

Configuration

Some bits might define global configuration settings, and be common to the Secure and Non-secure instances of the register.

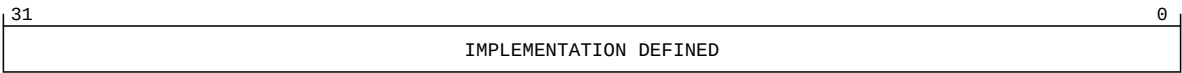
Attributes

ACTLR is a 32-bit register.

This register has the following instances:

- ACTLR.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TAC == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ACTLR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ACTLR();
end;
```


MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TAC == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        ACTLR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    ACTLR() = R(t);
end;
```

D10.2.2 ACTLR2, Auxiliary Control Register 2

The ACTLR2 characteristics are:

Purpose

Provides additional space to the ACTLR register to hold IMPLEMENTATION DEFINED trap functionality for execution at EL1 and EL0.

Configuration

In Armv8.0 and Armv8.1, it is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed. The implementation of this register can be detected by examining [ID_MMFR4.AC2](#).

From Armv8.2 this register must be implemented.

Attributes

ACTLR2 is a 32-bit register.

This register has the following instances:

- ACTLR2.

Field descriptions

31		0
IMPLEMENTATION DEFINED		

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR2

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TAC == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ACTLR2();
```

```
end;  
elseif PSTATE.EL == EL2 then  
    R(t) = ACTLR2();  
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b011

```
if PSTATE.EL == EL0 then  
    Undefined();  
elseif PSTATE.EL == EL1 then  
    if EL2Enabled() && HSTR().T1 == '1' then  
        AArch32_TakeHypTrapException(0x03);  
    elseif EL2Enabled() && HCR().TAC == '1' then  
        AArch32_TakeHypTrapException(0x03);  
    else  
        ACTLR2() = R(t);  
    end;  
elseif PSTATE.EL == EL2 then  
    ACTLR2() = R(t);  
end;
```

D10.2.3 ADFSR, Auxiliary Data Fault Status Register

The ADFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for Data Abort exceptions taken to EL1 modes.

Configuration

There are no configuration notes.

Attributes

ADFSR is a 32-bit register.

This register has the following instances:

- ADFSR.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ADFSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ADFSR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ADFSR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        ADFSR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    ADFSR() = R(t);
end;
```

D10.2.4 AIDR, Auxiliary ID Register

The AIDR characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED identification information.
The value of this register must be used in conjunction with the value of MIDR.

Configuration

There are no configuration notes.

Attributes

AIDR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing AIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = AIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = AIDR();
end;
```

D10.2.5 AIFSR, Auxiliary Instruction Fault Status Register

The AIFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for Prefetch Abort exceptions taken to EL1 modes.

Configuration

There are no configuration notes.

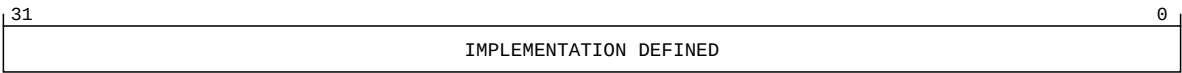
Attributes

AIFSR is a 32-bit register.

This register has the following instances:

- AIFSR.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AIFSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = AIFSR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = AIFSR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AIFSR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    AIFSR() = R(t);
end;
```


D10.2.6 AMAIR0, Auxiliary Memory Attribute Indirection Register 0

The AMAIR0 characteristics are:

Purpose

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR0.

Configuration

There are no configuration notes.

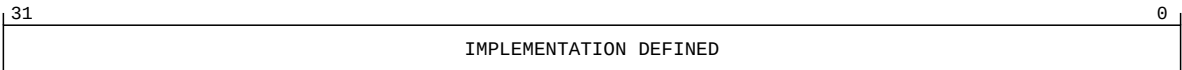
Attributes

AMAIR0 is a 32-bit register.

This register has the following instances:

- AMAIR0.

Field descriptions



This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = AMAIR0();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = AMAIR0();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AMAIR0() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    AMAIR0() = R(t);
end;
```

D10.2.7 AMAIR1, Auxiliary Memory Attribute Indirection Register 1

The AMAIR1 characteristics are:

Purpose

When using the Long-descriptor format translation tables for stage 1 translations, provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR1.

Configuration

When EL3 is using AArch32, write access to AMAIR1(S) is disabled when the CP15SDISABLE signal is asserted HIGH.

Attributes

AMAIR1 is a 32-bit register.

This register has the following instances:

- AMAIR1.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

This register is RES0 in the following cases:

- When an implementation does not provide any IMPLEMENTATION DEFINED memory attributes.
- When the Long-descriptor translation table format is not used.

Any IMPLEMENTATION DEFINED memory attributes are additional qualifiers for the memory locations and must not change the architected behavior specified by MAIR0 and MAIR1.

In a typical implementation, AMAIR0 and AMAIR1 split into eight one-byte fields, corresponding to the MAIRn.Attr<n> fields, but the architecture does not require them to do so.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = AMAIR1();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = AMAIR1();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AMAIR1() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    AMAIR1() = R(t);
end;
```

D10.2.8 APSR, Application Program Status Register

The APSR characteristics are:

Purpose

Hold program status and control information.

Note

Some of the fields in this register are permitted to return the value of the PSTATE field on a read. This is an exception to the general rule that an UNKNOWN field must not return information that cannot be obtained, at the current Privilege level, by an architected mechanism.

For more information see ‘[The Application Program Status Register, APSR](#)’.

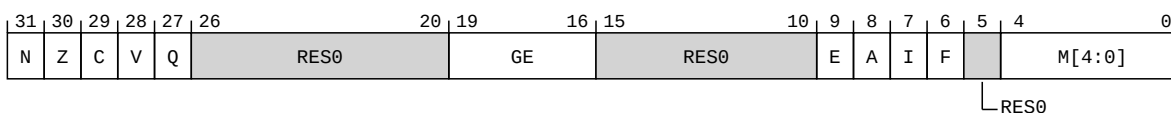
Configuration

This register is a Special-purpose register.

Attributes

APSR is a 32-bit register.

Field descriptions



N, bit [31]

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

Bits [26:20]

Reserved, RES0.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

Bits [15:10]

Reserved, RES0.

E, bit [9]

Endianness. This field is UNKNOWN, but is permitted to return the value of PSTATE.E field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.A field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.I field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. This field is UNKNOWN, but is permitted to return the value of PSTATE.F field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4:0], bits [4:0]

Mode. This field is UNKNOWN, but is permitted to return the value of PSTATE.M[4:0] field. On writes, this field is treated as Do-Not-Modify.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APSR

APSR can be read using the MRS instruction and written using the MSR (register) or MSR (immediate) instructions.

The MSR (register) and MRS instructions used to access APSR are data-independent-time instructions as described in About PSTATE.DIT.

D10.2.9 **ATS12NSOPR, Address Translate Stages 1 and 2 Non-secure Only PL1 Read**

The ATS12NSOPR characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if reading from the given virtual address.

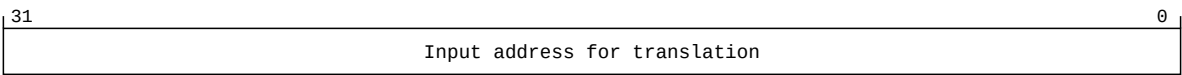
Configuration

There are no configuration notes.

Attributes

ATS12NSOPR is a 32-bit System instruction.

Field descriptions



IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing ATS12NSOPR

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_12, EL1, ATAccess_Read);
end;
```

D10.2.10 **ATS12NSOPW, Address Translate Stages 1 and 2 Non-secure Only PL1 Write**

The ATS12NSOPW characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL1 and the Non-secure state, with permissions as if writing to the given virtual address.

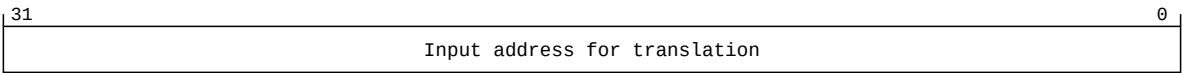
Configuration

There are no configuration notes.

Attributes

ATS12NSOPW is a 32-bit System instruction.

Field descriptions



IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing ATS12NSOPW

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_12, EL1, ATAccess_Write);
end;
```


D10.2.11 **ATS12NSOUR, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read**

The ATS12NSOUR characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if reading from the given virtual address.

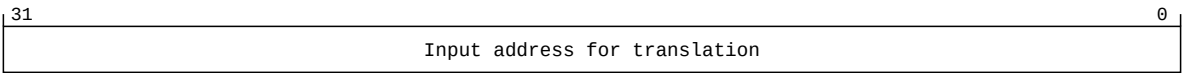
Configuration

There are no configuration notes.

Attributes

ATS12NSOUR is a 32-bit System instruction.

Field descriptions



IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing ATS12NSOUR

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_12, EL0, ATAccess_Read);
end;
```

D10.2.12 **ATS12NSOUW, Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write**

The ATS12NSOUW characteristics are:

Purpose

Performs stage 1 and 2 address translations as defined for PL0 and the Non-secure state, with permissions as if writing to the given virtual address.

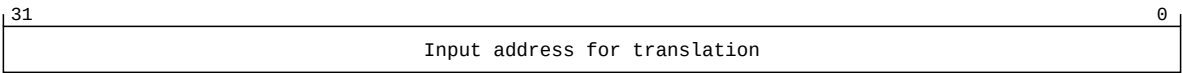
Configuration

There are no configuration notes.

Attributes

ATS12NSOUW is a 32-bit System instruction.

Field descriptions



IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the stage 2 translation.

Executing ATS12NSOUW

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_12, EL0, ATAccess_Write);
end;
```

D10.2.13 ATS1CPR, Address Translate Stage 1 Current state PL1 Read

The ATS1CPR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if reading from the given virtual address.

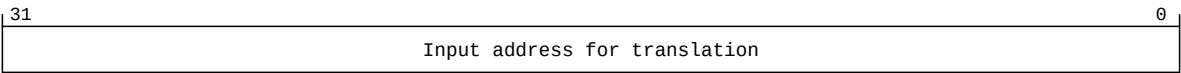
Configuration

There are no configuration notes.

Attributes

ATS1CPR is a 32-bit System instruction.

Field descriptions



IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CPR

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Read);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Read);
end;
```

D10.2.14 **ATS1CPW, Address Translate Stage 1 Current state PL1 Write**

The ATS1CPW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL1 and the current Security state, with permissions as if writing to the given virtual address.

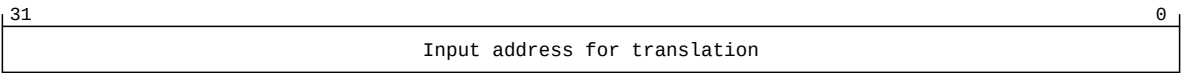
Configuration

There are no configuration notes.

Attributes

ATS1CPW is a 32-bit System instruction.

Field descriptions



IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CPW

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Write);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL1, ATAccess_Write);
end;
```

D10.2.15 **ATS1CUR, Address Translate Stage 1 Current state Unprivileged Read**

The ATS1CUR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if reading from the given virtual address.

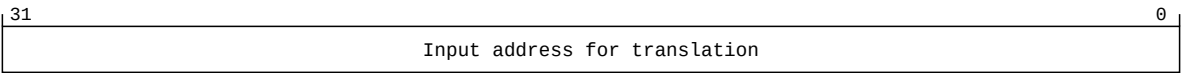
Configuration

There are no configuration notes.

Attributes

ATS1CUR is a 32-bit System instruction.

Field descriptions



IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CUR

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Read);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Read);
end;
```

D10.2.16 **ATS1CUW, Address Translate Stage 1 Current state Unprivileged Write**

The ATS1CUW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL0 and the current Security state, with permissions as if writing to the given virtual address.

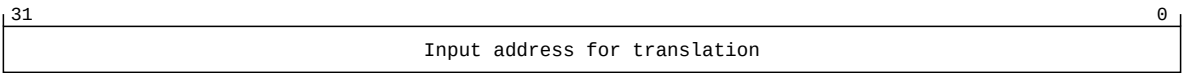
Configuration

There are no configuration notes.

Attributes

ATS1CUW is a 32-bit System instruction.

Field descriptions



IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. If EL2 is implemented and enabled in the current Security state, the resulting address is the IPA that is the output address of the stage 1 translation. Otherwise, the resulting address is a PA.

Executing ATS1CUW

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1000	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Write);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL0, ATAccess_Write);
end;
```

D10.2.17 **ATS1HR, Address Translate Stage 1 Hyp mode Read**

The ATS1HR characteristics are:

Purpose

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if reading from the given virtual address.

Configuration

There are no configuration notes.

Attributes

ATS1HR is a 32-bit System instruction.

Field descriptions

31	0
Input address for translation	

IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

Executing ATS1HR

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL2, ATAccess_Read);
end;
```

D10.2.18 **ATS1HW, Address Translate Stage 1 Hyp mode Write**

The ATS1HW characteristics are:

Purpose

Performs stage 1 address translation as defined for PL2 and the Non-secure state, with permissions as if writing to the given virtual address.

Configuration

There are no configuration notes.

Attributes

ATS1HW is a 32-bit System instruction.

Field descriptions

31		0
Input address for translation		

IA, bits [31:0]

Input address for translation. The resulting address can be read from the [PAR](#).

This System instruction takes a VA as input. The resulting address is the PA that is the output address of the translation.

Executing ATS1HW

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_AT(R(t), TranslationStage_1, EL2, ATAccess_Write);
end;
```


D10.2.19 BPIALL, Branch Predictor Invalidate All

The BPIALL characteristics are:

Purpose

Invalidate all entries from branch predictors.

Configuration

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

Attributes

BPIALL is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing BPIALL

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction. When [HCR.FB](#) is 1, at Non-secure EL1 this instruction executes as a [BPIALLIS](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b110

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().FB == '1' then
        BPIALLIS();
    else
        BPIALL();
    end;
elsif PSTATE.EL == EL2 then
    BPIALL();
end;
```

D10.2.20 BPIALLIS, Branch Predictor Invalidate All, Inner Shareable

The BPIALLIS characteristics are:

Purpose

Invalidate all entries from branch predictors Inner Shareable.

Configuration

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

Attributes

BPIALLIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing BPIALLIS

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction. Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0001	0b110

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        BPIALLIS();
    end;
elsif PSTATE.EL == EL2 then
    BPIALLIS();
end;
```

D10.2.21 BPIMVA, Branch Predictor Invalidate by VA

The BPIMVA characteristics are:

Purpose

Invalidate virtual address from branch predictors.

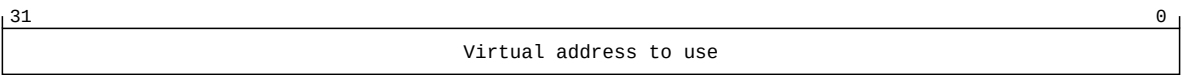
Configuration

In an implementation where the branch predictors are architecturally invisible, this instruction can execute as a NOP.

Attributes

BPIMVA is a 32-bit System instruction.

Field descriptions



VA, bits [31:0]

Virtual address to use.

Executing BPIMVA

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        BPIMVA(R(t));
    end;
elsif PSTATE.EL == EL2 then
    BPIMVA(R(t));
end;
```

D10.2.22 CCSIDR, Current Cache Size ID Register

The CCSIDR characteristics are:

Purpose

Provides information about the architecture of the currently selected cache.

When FEAT_CCIDX is implemented, this register is used in conjunction with CCSIDR2.

Configuration

The implementation includes one CCSIDR for each cache that it can access. [CSSELR](#) and the Security state select which Cache Size ID Register is accessible.

Attributes

CCSIDR is a 32-bit register.

Field descriptions

!IsFeatureImplemented(FEAT_CCIDX)

31	28	27	13	12	3	2	0
UNKNOWN		NumSets			Associativity		LineSize

Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [31:28]

Reserved, UNKNOWN.

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

($\log_2(\text{Number of bytes in cache line})$) - 4. For example:

For a line length of 16 bytes: $\log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.

For a line length of 32 bytes: $\log_2(32) = 5$, LineSize entry = 1.

Note

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.

Accessing CCSIDR

If [CSSELR](#).{Level, InD} is programmed to a cache level that is not implemented, then on a read of the CCSIDR the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR read is treated as NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID2 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CCSIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = CCSIDR();
end;
```

D10.2.23 CLIDR, Cache Level ID Register

The CLIDR characteristics are:

Purpose

Identifies the type of cache, or caches, that are implemented at each level and can be managed using the architected cache maintenance instructions that operate by set/way, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

Configuration

There are no configuration notes.

Attributes

CLIDR is a 32-bit register.

Field descriptions

31	30	29	27	26	24	23	21	20	18	17	15	14	12	11	9	8	6	5	3	2	0
ICB		LoUU		LoC		LoUIS		Ctype7	Ctype6	Ctype5	Ctype4	Ctype3	Ctype2	Ctype1							

ICB, bits [31:30]

Inner cache boundary. This field indicates the boundary for caching Inner Cacheable memory regions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ICB	Meaning
0b00	Not disclosed by this mechanism.
0b01	L1 cache is the highest Inner Cacheable level.
0b10	L2 cache is the highest Inner Cacheable level.
0b11	L3 cache is the highest Inner Cacheable level.

Access to this field is RO.

LoUU, bits [29:27]

Level of Unification Uniprocessor for the cache hierarchy.

For a description of the values of this field, see [Terminology for Clean, Invalidate, and Clean and Invalidate instructions](#).

Note
This field does not describe the requirements for instruction cache invalidation. See [CTR.DIC](#).

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

LoC, bits [26:24]

Level of Coherence for the cache hierarchy.

For a description of the values of this field, see [Terminology for Clean, Invalidate, and Clean and Invalidate instructions](#).

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

LoUIS, bits [23:21]

Level of Unification Inner Shareable for the cache hierarchy.

For a description of the values of this field, see [Terminology for Clean, Invalidate, and Clean and Invalidate instructions](#).

Note
This field does not describe the requirements for instruction cache invalidation. See [CTR.DIC](#).

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Ctype<n> , bits [3(n-1)+2:3(n-1)], for n = 7 to 1

Cache Type fields. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy.

Ctype<n>	Meaning
0b000	No cache.
0b001	Instruction cache only.
0b010	Data cache only.
0b011	Separate instruction and data caches.
0b100	Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of 0b000, no caches that can be managed using the architected cache maintenance instructions that operate by set/way exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of 0b000, the values of Ctype4 to Ctype7 must be ignored.

Accessing CLIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b001	0b0000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID2 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CLIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = CLIDR();
end;
```

D10.2.24 CONTEXTIDR, Context ID Register

The CONTEXTIDR characteristics are:

Purpose

Identifies the current Process Identifier and, when using the Short-descriptor translation table format, the Address Space Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

Configuration

The register format depends on whether address translation is using the Long-descriptor or the Short-descriptor translation table format.

Attributes

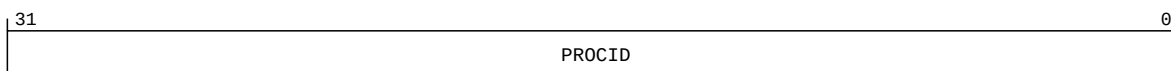
CONTEXTIDR is a 32-bit register.

This register has the following instances:

- CONTEXTIDR.

Field descriptions

TTBCR.EAE==1



PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CONTEXTIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    
```

```
        elsif EL2Enabled() && HCR().TRVM == '1' then
            AArch32_TakeHypTrapException(0x03);
        else
            R(t) = CONTEXTIDR();
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = CONTEXTIDR();
    end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        CONTEXTIDR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    CONTEXTIDR() = R(t);
end;
```

D10.2.25 CP15DMB, Data Memory Barrier System instruction

The CP15DMB characteristics are:

Purpose

Performs a Data Memory Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the DMB instruction instead.

Configuration

There are no configuration notes.

Attributes

CP15DMB is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing CP15DMB

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b101

```
if PSTATE.EL == EL0 then
    if SCTLRL().CP15BEN == '0' then
        Undefined();
    elseif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        CP15DMB();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif SCTLRL().CP15BEN == '0' then
        Undefined();
    else
        CP15DMB();
    end;
elseif PSTATE.EL == EL2 then
    if HSCTLRL().CP15BEN == '0' then
        Undefined();
    else
        CP15DMB();
    end;
end;
```

D10.2.26 CP15DSB, Data Synchronization Barrier System instruction

The CP15DSB characteristics are:

Purpose

Performs a Data Synchronization Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the DSB instruction instead.

Configuration

There are no configuration notes.

Attributes

CP15DSB is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing CP15DSB

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b100

```
if PSTATE.EL == EL0 then
    if SCTLR().CP15BEN == '0' then
        Undefined();
    elseif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        CP15DSB();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif SCTLR().CP15BEN == '0' then
        Undefined();
    else
        CP15DSB();
    end;
elseif PSTATE.EL == EL2 then
    if HSCTLR().CP15BEN == '0' then
        Undefined();
    else
        CP15DSB();
    end;
end;
```

D10.2.27 CP15ISB, Instruction Synchronization Barrier System instruction

The CP15ISB characteristics are:

Purpose

Performs an Instruction Synchronization Barrier.

Arm deprecates any use of this System instruction, and strongly recommends that software use the ISB instruction instead.

Configuration

There are no configuration notes.

Attributes

CP15ISB is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing CP15ISB

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b100

```
if PSTATE.EL == EL0 then
    if SCTLRL().CP15BEN == '0' then
        Undefined();
    elseif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        CP15ISB();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif SCTLRL().CP15BEN == '0' then
        Undefined();
    else
        CP15ISB();
    end;
elseif PSTATE.EL == EL2 then
    if HSCTLRL().CP15BEN == '0' then
        Undefined();
    else
        CP15ISB();
    end;
end;
```

D10.2.28 CPACR, Architectural Feature Access Control Register

The CPACR characteristics are:

Purpose

Controls access to trace, and to Advanced SIMD and floating-point functionality from EL0, EL1, and EL3.

In an implementation that includes EL2, the CPACR has no effect on instructions executed at EL2.

Configuration

Bits in the **NSACR** control Non-secure access to the CPACR fields. See the field descriptions for more information.

Note

In the register field descriptions, controls are described as applying at specified Privilege levels. This is because, in Secure state, a PL1 control:

- Applies to execution in a Secure EL3 mode when EL3 is using AArch32.
- Applies to execution in a Secure EL1 mode when EL3 is using AArch64.

See ‘Security state, Exception levels, and AArch32 execution privilege’.

Attributes

CPACR is a 32-bit register.

Field descriptions

**ASEDIS, bit [31]**

Disables PL0 and PL1 execution of Advanced SIMD instructions.

ASEDIS	Meaning
0b0	This control permits execution of Advanced SIMD instructions at PL0 and PL1.
0b1	All instruction encodings that are Advanced SIMD instruction encodings, but are not also floating-point instruction encodings, are UNDEFINED at PL0 and PL1.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

For the list of instructions affected by this field, see ‘Controls of Advanced SIMD operation that do not apply to floating-point operation’.

See the description of CPACR.cp10 for a list of other controls that can disable or trap execution of Advanced SIMD instructions in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [30:29]

Reserved, RES0.

TRCDIS, bit [28]

Traps PL0 and PL1 System register accesses to all implemented trace System registers to Undefined mode.

TRCDIS	Meaning
0b0	This control has no effect on PL0 and PL1 System register accesses to trace System registers.
0b1	PL0 and PL1 System register accesses to all implemented trace System registers are trapped to Undefined mode.

If the implementation does not include a trace unit, or does not include a System register interface to the trace unit registers, this field is RES0. Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, it is RAZ/WI.

Note

- The ETMv4 architecture does not permit EL0 to access the trace System registers. If the trace unit implements FEAT_ETMv4, EL0 accesses to the trace System registers are UNDEFINED.
- The architecture does not provide traps on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:24]

Reserved, RES0.

cp11, bits [23:22]

The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the CPACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

cp10, bits [21:20]

Defines the access rights for the Advanced SIMD and floating-point functionality. Possible values of the field are:

cp10	Meaning
0b00	PL0 and PL1 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.
0b01	PL0 accesses to Advanced SIMD and floating-point registers or instructions are UNDEFINED.

cp10	Meaning
0b10	Reserved. The effect of programming this field to this value is CONSTRAINED UNPREDICTABLE choice between: - The value of this field is treated as 0b00, 0b01, or 0b11 for all purposes other than reading the value of the field. - PL1 accesses to floating-point and Advanced SIMD registers or instructions are UNDEFINED.
0b11	This control permits full access to the Advanced SIMD and floating-point functionality from PL0 and PL1.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the [FPSCR](#), [FPSID](#), [MVFR0](#), [MVFR1](#), [MVFR2](#), or [FPEXC](#) System registers.

Note

The [CPACR](#) has no effect on Advanced SIMD and floating-point accesses from PL2. These can be disabled by the [HCPTR.TCP10](#) field.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

Execution of Advanced SIMD and floating-point instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10, or, if executing at EL0, CPACR_EL1.FPEN.
- [FPEXC](#).EN.
- If executing in Non-secure state:
 - [HCPTR.TCP10](#), or if EL2 is using AArch64, CPTR_EL2.TFP.
 - [NSACR](#).cp10, or if EL3 is using AArch64, CPTR_EL3.TFP.
- For Advanced SIMD instructions only:
 - CPACR.ASEDIS.
 - If executing in Non-secure state, [HCPTR.TASE](#) and [NSACR.NSASEDIS](#).

See the descriptions of the controls for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

Bits [19:0]

Reserved, RES0.

Accessing CPACR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b010


```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCPTR().TCPAC == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CPACR();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = CPACR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCPTR().TCPAC == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        CPACR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    CPACR() = R(t);
end;
```

D10.2.29 CPSR, Current Program Status Register

The CPSR characteristics are:

Purpose

Holds PE status and control information.

Configuration

This register is a Special-purpose register.

Attributes

CPSR is a 32-bit register.

Field descriptions

31	30	29	28	27	26	20				19	16				15	10				9	8	7	6	5	4	3	0	
N	Z	C	V	Q	RES0				GE				RES0				E	A	I	F	RES0		RES1		M			

N, bit [31]

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

Bits [26:20]

Reserved, RES0.

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

Bits [15:10]

Reserved, RES0.

E, bit [9]

Endianness state bit. Controls the load and store endianness for data accesses:

E	Meaning
0b0	Little-endian operation
0b1	Big-endian operation.

Instruction fetches ignore this bit.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any Exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any Exception level other than EL0.

When the reset value of the [SCTLR.EE](#) bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

A, bit [8]

SError exception mask bit.

A	Meaning
0b0	Exception not masked.
0b1	Exception masked.

I, bit [7]

IRQ mask bit.

I	Meaning
0b0	Exception not masked.
0b1	Exception masked.

F, bit [6]

FIQ mask bit.

F	Meaning
0b0	Exception not masked.
0b1	Exception masked.

Bit [5]

Reserved, RES0.

Bit [4]

Reserved, RES1.

M, bits [3:0]

Current PE mode.

M	Meaning
0b0000	User.
0b0001	FIQ.
0b0010	IRQ.
0b0011	Supervisor.
0b0110	Monitor.
0b0111	Abort.
0b1010	Hyp.
0b1011	Undefined.
0b1111	System.

Accessing CPSR

CPSR can be read using the MRS instruction and written using the MSR (register) or MSR (immediate) instructions.

The MSR (register) and MRS instructions used to access CPSR are data-independent-time instructions as described in About PSTATE.DIT.

D10.2.30 CSSELR, Cache Size Selection Register

The CSSELR characteristics are:

Purpose

Selects the current Cache Size ID Register, [CCSIDR](#), by specifying the required cache level and the cache type, which is either instruction cache or data cache.

If FEAT_CCIDX is implemented, CSSELR also selects the current CCSIDR2.

Configuration

There are no configuration notes.

Attributes

CSSELR is a 32-bit register.

This register has the following instances:

- CSSELR.

Field descriptions



Bits [31:4]

Reserved, RES0.

Level, bits [3:1]

Cache level of required cache. Permitted values are:

Level	Meaning
0b000	Level 1 cache.
0b001	Level 2 cache.
0b010	Level 3 cache.
0b011	Level 4 cache.
0b100	Level 5 cache.
0b101	Level 6 cache.
0b110	Level 7 cache.

All other values are reserved.

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

InD, bit [0]

Instruction not Data bit. Permitted values are:

InD	Meaning
0b0	Data or unified cache.
0b1	Instruction cache.

If CSSELR.{Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CSSELR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b010	0b0000	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID2 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CSSELR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = CSSELR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b010	0b0000	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID2 == '1' then

```

```
        AArch32_TakeHypTrapException(0x03);  
    else  
        CSSELR() = R(t);  
    end;  
elseif PSTATE.EL == EL2 then  
    CSSELR() = R(t);  
end;
```

D10.2.31 CTR, Cache Type Register

The CTR characteristics are:

Purpose

Provides information about the architecture of the caches.

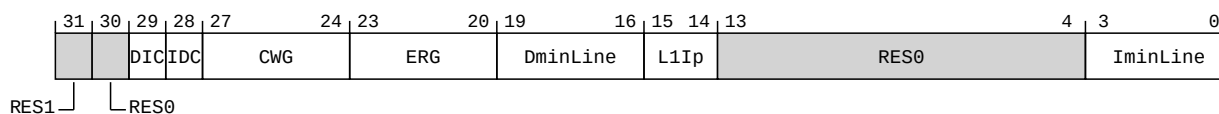
Configuration

There are no configuration notes.

Attributes

CTR is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES1.

Bit [30]

Reserved, RES0.

DIC, bit [29]

Instruction cache invalidation requirements for data to instruction coherence.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DIC	Meaning
0b0	Instruction cache invalidation to the Point of Unification is required for data to instruction coherence.
0b1	Instruction cache invalidation to the Point of Unification is not required for data to instruction coherence.

All PEs in the same Inner Shareable shareability domain must have a common value of this field.

Access to this field is RO.

IDC, bit [28]

Data cache clean requirements for instruction to data coherence. The meaning of this bit is:

The value of this field is an IMPLEMENTATION DEFINED choice of:

IDC	Meaning
0b0	Data cache clean to the Point of Unification is required for instruction to data coherence, unless CLIDR.LoC == 0b000 or (CLIDR.LoUIS == 0b000 and CLIDR.LoUU == 0b000).
0b1	Data cache clean to the Point of Unification is not required for instruction to data coherence.

If CTR.DIC is 1, then the value reported in this field must be 1.

The Effective value of IDC is 1 if any of the following are true:

- CTR.IDC == 1.
- CLIDR.LoC == 0b000.
- CLIDR.LoUIS == 0b000 and CLIDR.LoUU == 0b000.

All PEs in the same Inner Shareable shareability domain must have a common Effective value of IDC.

Access to this field is RO.

CWG, bits [27:24]

Cache writeback granule. $\log_2$ of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of 0b0000 indicates that this register does not provide Cache writeback granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache writeback granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than 0b1001 are reserved.

Arm recommends that an implementation that does not support cache write-back implements this field as 0b0001. This applies, for example, to an implementation that supports only write-through caches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

ERG, bits [23:20]

Exclusives reservation granule. $\log_2$ of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The use of the value 0b0000 is deprecated.

The value 0b0001 and values greater than 0b1001 are reserved.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

DminLine, bits [19:16]

$\log_2$ of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

L1Ip, bits [15:14]

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Ip	Meaning
0b00	Reserved.
0b01	ASID-tagged Virtual Index, Virtual Tag (AIVIVT).
0b10	Virtual Index, Physical Tag (VIPT).
0b11	Physical Index, Physical Tag (PIPT).

The table above shows all possible values of this field. The values this field can take in the current implementation are:

- 0b10
- 0b11

From Armv8.0, the value 0b01 is not permitted.

Access to this field is RO.

Bits [13:4]

Reserved, RES0.

IminLine, bits [3:0]

Log₂ of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID2 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CTR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = CTR();
end;
```

D10.2.32 DCCIMVAC, Data Cache line Clean and Invalidate by VA to PoC

The DCCIMVAC characteristics are:

Purpose

Clean and Invalidate data or unified cache line by virtual address to PoC.

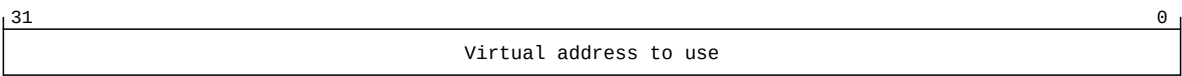
Configuration

There are no configuration notes.

Attributes

DCCIMVAC is a 32-bit System instruction.

Field descriptions



VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DCCIMVAC

Execution of this instruction might require an address translation from VA to PA, and that translation might fault.

For more information about faults, see [‘Permission fault’](#).

For more information about data cache maintenance instructions, see [‘AArch32 data cache maintenance instructions \(DC*\)’](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1110	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapDC(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TPC == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatDCAsNOP(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_PoC);
        end;
    end;
```

```
        end;
    elsif PSTATE.EL == EL2 then
        if !AArch32_CanTrapDC(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            if AArch32_TreatDCAsNOP(CacheOp_CleanInvalidate, CacheOpScope_PoC) then
                ExecuteAsNOP();
            else
                AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_PoC);
            end;
        end;
    end;
end;
```

D10.2.33 DCCISW, Data Cache line Clean and Invalidate by Set/Way

The DCCISW characteristics are:

Purpose

Clean and Invalidate data or unified cache line by set/way.

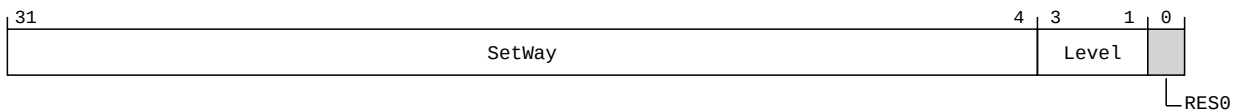
Configuration

There are no configuration notes.

Attributes

DCCISW is a 32-bit System instruction.

Field descriptions



SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

$A = \log_2(\text{ASSOCIATIVITY})$, $L = \log_2(\text{LINELEN})$, $B = (L + S)$, $S = \log_2(\text{NSETS})$.

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DCCISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONstrained UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
 - No cache lines.
 - A single arbitrary cache line.
 - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1110	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TSW == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_SetWay);
    end;
elseif PSTATE.EL == EL2 then
    AArch32_DC(R(t), CacheOp_CleanInvalidate, CacheOpScope_SetWay);
end;
```

D10.2.34 DCCMVAC, Data Cache line Clean by VA to PoC

The DCCMVAC characteristics are:

Purpose

Clean data or unified cache line by virtual address to PoC.

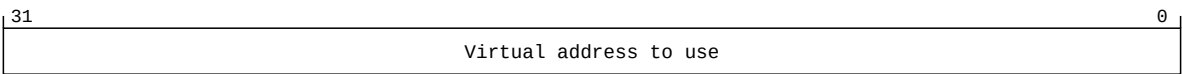
Configuration

There are no configuration notes.

Attributes

DCCMVAC is a 32-bit System instruction.

Field descriptions



VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DCCMVAC

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see '[AArch32 data cache maintenance instructions \(DC*\)](#)'.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if !AArch32_CanTrapDC(CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elseif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TPC == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatDCAsNOP(CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
elseif PSTATE.EL == EL2 then
    if !AArch32_CanTrapDC(CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
```

```
    else
        if AArch32_TreatDCasNOP(CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoC);
        end;
    end;
end;
```

D10.2.35 DCCMVAU, Data Cache line Clean by VA to PoU

The DCCMVAU characteristics are:

Purpose

Clean data or unified cache line by virtual address to PoU.

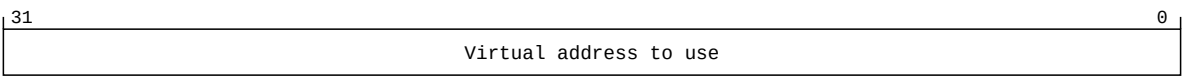
Configuration

There are no configuration notes.

Attributes

DCCMVAU is a 32-bit System instruction.

Field descriptions



VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DCCMVAU

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see '[AArch32 data cache maintenance instructions \(DC*\)](#)'.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1011	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TPU == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoU);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_PoU);
end;
```

D10.2.36 DCCSW, Data Cache line Clean by Set/Way

The DCCSW characteristics are:

Purpose

Clean data or unified cache line by set/way.

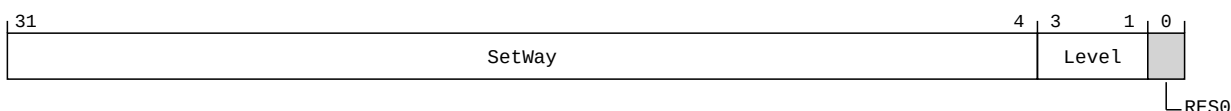
Configuration

There are no configuration notes.

Attributes

DCCSW is a 32-bit System instruction.

Field descriptions



SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

$A = \log_2(\text{ASSOCIATIVITY})$, $L = \log_2(\text{LINELEN})$, $B = (L + S)$, $S = \log_2(\text{NSETS})$.

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DCCSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONstrained UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
 - No cache lines.
 - A single arbitrary cache line.
 - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b1010	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TSW == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_SetWay);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DC(R(t), CacheOp_Clean, CacheOpScope_SetWay);
end;
```

D10.2.37 DCIMVAC, Data Cache line Invalidate by VA to PoC

The DCIMVAC characteristics are:

Purpose

Invalidate data or unified cache line by virtual address to PoC.

Configuration

There are no configuration notes.

Attributes

DCIMVAC is a 32-bit System instruction.

Field descriptions

31	0
Virtual address to use	

VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DCIMVAC

It is IMPLEMENTATION DEFINED whether, when this instruction is executed, it can generate a watchpoint. If this instruction can generate a watchpoint this is prioritized in the same way as other watchpoints.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see '[AArch32 data cache maintenance instructions \(DC*\)](#)'.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0110	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapDC(CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TPC == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatDCAsNOP(CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
end;

```

```
elseif PSTATE.EL == EL2 then
    if !AArch32_CanTrapDC(CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch32_TreatDCAsNOP(CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_PoC);
        end;
    end;
end;
```

D10.2.38 DCISW, Data Cache line Invalidate by Set/Way

The DCISW characteristics are:

Purpose

Invalidate data or unified cache line by set/way.

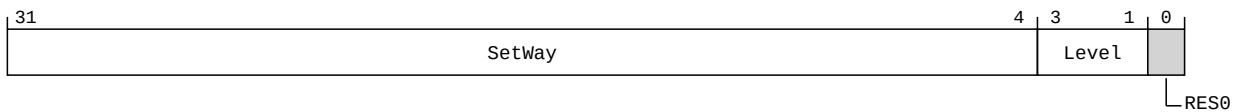
Configuration

There are no configuration notes.

Attributes

DCISW is a 32-bit System instruction.

Field descriptions



SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

$A = \log_2(\text{ASSOCIATIVITY})$, $L = \log_2(\text{LINELEN})$, $B = (L + S)$, $S = \log_2(\text{NSETS})$.

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DCISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONstrained UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED
- The instruction performs cache maintenance on one of:
 - No cache lines.
 - A single arbitrary cache line.
 - Multiple arbitrary cache lines.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0110	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TSW == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_SetWay);
    end;
elseif PSTATE.EL == EL2 then
    AArch32_DC(R(t), CacheOp_Invalidate, CacheOpScope_SetWay);
end;
```

D10.2.39 DFAR, Data Fault Address Register

The DFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception.

Configuration

There are no configuration notes.

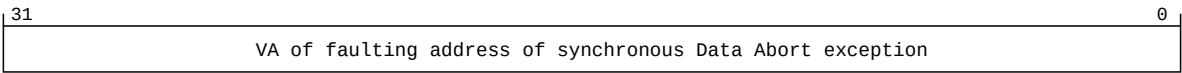
Attributes

DFAR is a 32-bit register.

This register has the following instances:

- DFAR.

Field descriptions



VA, bits [31:0]

VA of faulting address of synchronous Data Abort exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DFAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = DFAR();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = DFAR();
end;
```


MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        DFAR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    DFAR() = R(t);
end;
```

D10.2.40 **DFSR, Data Fault Status Register**

The DFSR characteristics are:

Purpose

Holds status information about the last data fault.

Configuration

The current translation table format determines which format of the register is used.

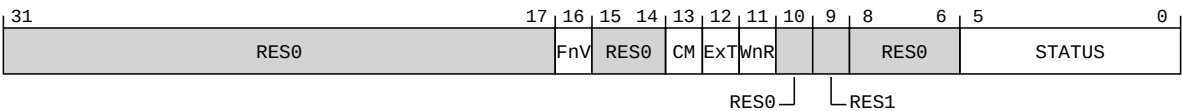
Attributes

DFSR is a 32-bit register.

This register has the following instances:

- DFSR.

Field descriptions



Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort.

FnV	Meaning
0b0	DFAR is valid.
0b1	DFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort. It is RES0 for all other Data Abort exceptions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:14]

Reserved, RES0.

CM, bit [13]

Cache maintenance fault. For synchronous faults, this bit indicates whether a cache maintenance instruction generated the fault.

CM	Meaning
0b0	Abort not caused by execution of a cache maintenance instruction.
0b1	Abort caused by execution of a cache maintenance instruction.

On an asynchronous fault, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ExT, bit [12]

External abort type.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [11]

Write not Read bit. Indicates whether the abort was caused by a write or a read instruction.

WnR	Meaning
0b0	Abort caused by a read instruction.
0b1	Abort caused by a write instruction.

For faults on the cache maintenance and address translation System instructions in the (coproc==0b1111) encoding space this bit always returns a value of 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [10]

Reserved, RES0.

Bit [9]

Reserved, RES1.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

STATUS	Meaning
0b000100	Translation fault.
0b001100	Permission fault.
0b010000	Synchronous External abort, other than synchronous parity or ECC error.
0b010001	Asynchronous SError exception.
0b011000	Synchronous parity or ECC error on memory access.
0b011001	Asynchronous SError exception, from a parity or ECC error on memory access.
0b100001	Alignment fault.
0b100010	Debug exception.
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive access).

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DFSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = DFSR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DFSR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        DFSR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    DFSR() = R(t);
end;
```

D10.2.41 DTLBIALL, Data TLB Invalidate All

The DTLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
 - Would be required for the EL1&0 translation regime.
 - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the Non-secure PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This instruction is treated as a NOP.

Attributes

DTLBIALL is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing DTLBIALL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DTLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
    end;
elseif PSTATE.EL == EL2 then
    AArch32_DTLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
end;
```

D10.2.42 DTLBIASID, Data TLB Invalidate by ASID match

The DTLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
 - Is from a level of lookup above the final level.
 - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This instruction is treated as a NOP.

Attributes

DTLBIASID is a 32-bit System instruction.

Field descriptions



Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing DTLBIASID

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DTLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
            ↪R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_DTLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr, R(t));
end;
```


D10.2.43 DTLBIMVA, Data TLB Invalidate by VA

The DTLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from data TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
 - The entry is from a level of lookup above the final level and matches the specified ASID.
 - The entry is a global entry from the final level of lookup.
 - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

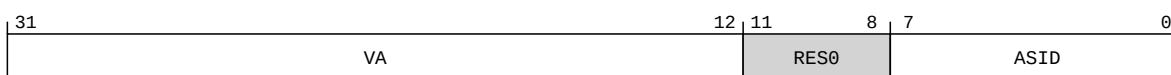
Configuration

This instruction is treated as a NOP.

Attributes

DTLBIMVA is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Executing DTLBIMVA

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_DTLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_DTLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```

D10.2.44 ELR_hyp, Exception Link Register (Hyp mode)

The ELR_hyp characteristics are:

Purpose

When taking an exception to Hyp mode, holds the address to return to.

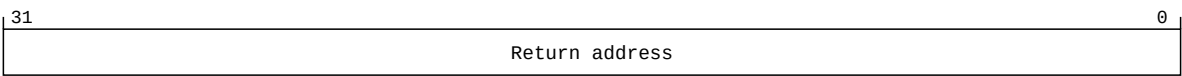
Configuration

This register is a Special-purpose register.

Attributes

ELR_hyp is a 32-bit register.

Field descriptions



ADDR, bits [31:0]

Return address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_hyp

ELR_hyp is accessible only at Hyp mode and Monitor mode.

Accesses to this register use the following encodings in the System register encoding space:

MRS{<c>}{<q>} <Rd>, ELR_hyp

R	M	M1
0b0	0b1	0b1110

MSR{<c>}{<q>} ELR_hyp, <Rn>

R	M	M1
0b0	0b1	0b1110

D10.2.45 FCSEIDR, FCSE Process ID register

The FCSEIDR characteristics are:

Purpose

Identifies whether the Fast Context Switch Extension (FCSE) is implemented.

From Armv8.0, the FCSE is not implemented, so this register is RAZ/WI. Software can access this register to determine that the implementation does not include the FCSE.

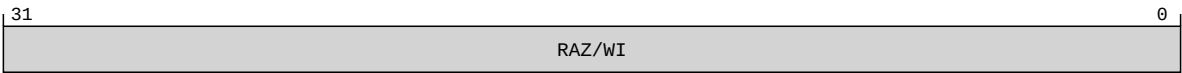
Configuration

There are no configuration notes.

Attributes

FCSEIDR is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RAZ/WI.

Accessing FCSEIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = FCSEIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = FCSEIDR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        FCSEIDR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    FCSEIDR() = R(t);
end;
```

D10.2.46 FPEXC, Floating-Point Exception Control register

The FPEXC characteristics are:

Purpose

Provides a global enable for the implemented Advanced SIMD and floating-point functionality, and reports floating-point status information.

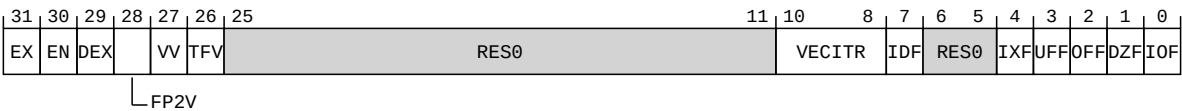
Configuration

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

Attributes

FPEXC is a 32-bit register.

Field descriptions



EX, bit [31]

Exception bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RAZ/WI.

EN, bit [30]

Enables access to the Advanced SIMD and floating-point functionality from all Exception levels, except that setting this field to 0 does not disable the following:

- VMSR accesses to the FPEXC or FPSID.
- VMRS accesses from the FPEXC, FPSID, MVFR0, MVFR1, or MVFR2.

EN	Meaning
0b0	Accesses to the FPSCR, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers, are UNDEFINED at all Exception levels.
0b1	This control permits access to the Advanced SIMD and floating-point functionality at all Exception levels.

Execution of Advanced SIMD and floating-point instructions in AArch32 state can be disabled or trapped by the following controls:

- CPACR.cp10.
- FPEXC.EN.
- If executing in Non-secure state:
 - HCPTR.TCP10.
 - NSACR.cp10.

- For Advanced SIMD instructions only:
 - CPACR.ASEDIS.
 - If executing in Non-secure state, [HCPTR.TASE](#) and [NSACR.NSASEDIS](#).

See the descriptions of the controls for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

DEX, bit [29]

Defined synchronous exception on floating-point execution.

This field identifies whether a synchronous exception generated by the attempted execution of an instruction was generated by an unallocated encoding. The instruction must be in the encoding space that is identified by the pseudocode function `ExecutingCP10or11Instr()` returning TRUE. This field also indicates whether the FPEXC.TFV field is valid.

The meaning of this bit is:

DEX	Meaning
0b0	The exception was generated by the attempted execution of an unallocated instruction in the encoding space that is identified by the pseudocode function <code>ExecutingCP10or11Instr()</code> . If FPEXC.TFV is RW then it is invalid and UNKNOWN. If FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.
0b1	The exception was generated during the execution of an allocated encoding. FPEXC.TFV is valid and indicates the cause of the exception.

On an exception that sets this bit to 1 the exception-handling routine must clear this bit to 0.

On an implementation that both does not support trapping of floating-point exceptions and implements the [FPSCR](#).{Stride, Len} fields as RAZ, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FP2V, bit [28]

FPINST2 instruction Valid bit. From Armv8.0, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES0.

VV, bit [27]

VECITR Valid bit. From Armv8.0, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES0.

TFV, bit [26]

Trapped Fault Valid bit. Valid only when the value of FPEXC.DEX is 1. When valid, it indicates the cause of the exception and therefore whether FPEXC.{IDF, IXF, UFF, OFF, DZF, IOF} are valid.

TFV	Meaning
0b0	The exception was caused by the execution of a floating-point VABS, VADD, VDIV, VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VMOV, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VSQRT, or VSUB instruction when one or both of FPSCR .{Stride, Len} was nonzero. If FPEXC .{IDF, IXF, UFF, OFF, DZF, IOF} are RW then they are invalid and UNKNOWN.
0b1	FPEXC .{IDF, IXF, UFF, OFF, DZF, IOF} indicate the presence of trapped floating-point exceptions that had occurred at the time of the exception. Bits are set for all trapped exceptions that had occurred at the time of the exception.

This bit returns a status value and ignores writes.

When the value of [FPEXC](#).DEX is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When an implementation does not implement trapping of floating-point exceptions, access to this field is RAZ/WI.
- When an implementation implements [FPSCR](#).{Stride, Len} as RAZ, access to this field is RAO/WI.

Bits [25:11]

Reserved, RES0.

VECITR, bits [10:8]

Vector Iteration count. From Armv8.0, this field is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RES1.

IDF, bit [7]

Input Denormal trapped exception bit. Valid only when the value of [FPEXC](#).TFV is 1. When valid, it indicates whether an Input Denormal exception occurred while [FPSCR](#).IDE was 1:

IDF	Meaning
0b0	Input Denormal exception has not occurred.
0b1	Input Denormal exception has occurred.

Input Denormal exceptions can occur only when [FPSCR](#).FZ is 1.

Note

A half-precision floating-point value that is flushed to zero because the value of [FPSCR](#).FZ16 is 1 does not generate an Input Denormal exception.

This bit must be cleared to 0 by the exception-handling routine.

When the value of [FPEXC](#).TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Input Denormal floating-point exceptions, access to this field is RAZ/WI.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Inexact exception occurred while [FPSCR.IXE](#) was 1:

IXF	Meaning
0b0	Inexact exception has not occurred.
0b1	Inexact exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Inexact floating-point exceptions, access to this field is RAZ/WI.

UFF, bit [3]

Underflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Underflow exception occurred while [FPSCR.UFE](#) was 1:

UFF	Meaning
0b0	Underflow exception has not occurred.
0b1	Underflow exception has occurred.

Underflow trapped exceptions can occur:

- On half-precision data-processing instructions only when [FPSCR.FZ16](#) is 0.
- Otherwise only when [FPSCR.FZ](#) is 0.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Underflow floating-point exceptions, access to this field is RAZ/WI.

OFF, bit [2]

Overflow trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Overflow exception occurred while [FPSCR.OFE](#) was 1:

OFF	Meaning
0b0	Overflow exception has not occurred.
0b1	Overflow exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Overflow floating-point exceptions, access to this field is RAZ/WI.

DZE, bit [1]

Divide by Zero trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether a Divide by Zero exception occurred while [FPSCR.DZE](#) was 1:

DZF	Meaning
0b0	Divide by Zero exception has not occurred.
0b1	Divide by Zero exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Divide by Zero floating-point exceptions, access to this field is RAZ/WI.

IOF, bit [0]

Invalid Operation trapped exception bit. Valid only when the value of FPEXC.TFV is 1. When valid, it indicates whether an Invalid Operation exception occurred while [FPSCR.IOE](#) was 1:

IOF	Meaning
0b0	Invalid Operation exception has not occurred.
0b1	Invalid Operation exception has occurred.

This bit must be cleared to 0 by the exception-handling routine.

When the value of FPEXC.TFV is 0 and this bit is RW, this bit is invalid and UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Invalid Operation floating-point exceptions, access to this field is RAZ/WI.

Accessing FPEXC

Accesses to this register use the following encodings in the System register encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg
0b1000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        R(t) = FPEXC();
    end;
elsif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        R(t) = FPEXC();
    end;
end;
```

VMSR{<c>}{<q>} <spec_reg>, <Rt>

reg
0b1000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        FPEXC() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        FPEXC() = R(t);
    end;
end;
```

D10.2.47 FPSCR, Floating-Point Status and Control Register

The FPSCR characteristics are:

Purpose

Provides floating-point system status information and control.

Configuration

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to nonzero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

Attributes

FPSCR is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	Z	C	V	QC	AHP	DN	FZ	RMode	Stride				Len	IDE	RES0	IXE	UFE	OFE	DZE	IOE	IDC	RES0	IXC	UFC	OFC	DZC	IOC			

└─ RES0

N, bit [31]

Negative condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow condition flag. This is updated by floating-point comparison operations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AHP, bit [26]

Alternative half-precision control bit:

AHP	Meaning
0b0	IEEE half-precision format selected.
0b1	Alternative half-precision format selected.

This bit is used only for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the FEAT_FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DN, bit [25]

Default NaN control bit:

DN	Meaning
0b0	NaN operands propagate through to the output of a floating-point operation.
0b1	Any operation involving one or more NaNs returns the default NaN.

The value of this bit controls only scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the default NaN setting, regardless of the value of the DN bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FZ, bit [24]

Flush-to-zero mode control bit:

FZ	Meaning
0b0	Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.
0b1	Flush-to-zero mode enabled.

The value of this bit controls only scalar floating-point arithmetic. Advanced SIMD arithmetic always uses the Flush-to-zero setting, regardless of the value of the FZ bit.

This bit has no effect on half-precision calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field. The encoding of this field is:

RMode	Meaning
0b00	Round to Nearest (RN) mode.
0b01	Round towards Plus Infinity (RP) mode.
0b10	Round towards Minus Infinity (RM) mode.
0b11	Round towards Zero (RZ) mode.

The specified rounding mode is used by almost all scalar floating-point instructions. Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

Arm strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.{Stride, Len} as RAZ, access to this field is RAZ/WI.

Bit [19]

Reserved, RES0.

Len, bits [18:16]

If this field is RW and is set to a value other than zero, some floating-point instruction encodings are UNDEFINED. The instruction pseudocode identifies these instructions.

Arm strongly recommends that software never sets this field to a value other than zero.

The value of this field is ignored when processing Advanced SIMD instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.{Stride, Len} as RAZ, access to this field is RAZ/WI.

IDE, bit [15]

Input Denormal floating-point exception trap enable.

IDE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the IDC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IDC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Input Denormal floating-point exceptions, access to this field is RAZ/WI.

Bits [14:13]

Reserved, RES0.

IXE, bit [12]

Inexact floating-point exception trap enable.

IXE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the IXC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IXC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Inexact floating-point exceptions, access to this field is RAZ/WI.

UFE, bit [11]

Underflow floating-point exception trap enable.

UFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the UFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs and Flush-to-zero is not enabled, the PE does not update the UFC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Underflow floating-point exceptions, access to this field is RAZ/WI.

OFE, bit [10]

Overflow floating-point exception trap enable.

OFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the OFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the OFC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Overflow floating-point exceptions, access to this field is RAZ/WI.

DZE, bit [9]

Divide by Zero floating-point exception trap enable.

DZE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the DZC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the DZC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Divide by Zero floating-point exceptions, access to this field is RAZ/WI.

IOE, bit [8]

Invalid Operation floating-point exception trap enable.

IOE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the IOC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the IOC bit.

When this bit is RW, it applies only to floating-point operations. Advanced SIMD operations always use untrapped floating-point exception handling in AArch32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Invalid Operation floating-point exceptions, access to this field is RAZ/WI.

IDC, bit [7]

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IDE bit.

Advanced SIMD instructions set this bit if the Input Denormal floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IDE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXC, bit [4]

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IXE bit.

Advanced SIMD instructions set this bit if the Inexact floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IXE bit.

The criteria for the Inexact floating-point exception to occur are different in Flush-to-zero mode. For more information, see ‘Flush-to-zero’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFC, bit [3]

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the UFE bit.

Advanced SIMD instructions set this bit if the Underflow floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, if FPSCR.UFE is 0 or if Flush-to-zero is enabled.

The criteria for the Underflow floating-point exception to occur are different in Flush-to-zero mode. For more information, see ‘Flush-to-zero’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFC, bit [2]

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the OFE bit.

Advanced SIMD instructions set this bit if the Overflow floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the OFE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZC, bit [1]

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the DZE bit.

Advanced SIMD instructions set this bit if the Divide by Zero floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the DZE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOC, bit [0]

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How VFP instructions update this bit depends on the value of the IOE bit.

Advanced SIMD instructions set this bit if the Invalid Operation floating-point exception occurs in one or more of the floating-point calculations performed by the instruction, regardless of the value of the IOE bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FPSCR

Accesses to this register use the following encodings in the System register encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg

0b0001

```
if PSTATE.EL == EL0 then
    if CPACR().cp10 IN {'0x'} then
        Undefined();
    elseif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        R(t) = FPSCR();
    end;
elseif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
        Undefined();
    elseif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        R(t) = FPSCR();
    end;
elseif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        R(t) = FPSCR();
    end;
end;
```

VMSR{<c>}{<q>} <spec_reg>, <Rt>

reg
0b0001

```
if PSTATE.EL == EL0 then
    if CPACR().cp10 IN {'0x'} then
        Undefined();
    elseif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        FPSCR() = R(t);
    end;
elseif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
        Undefined();
    elseif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        FPSCR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        FPSCR() = R(t);
    end;
end;
```

D10.2.48 FPSID, Floating-Point System ID register

The FPSID characteristics are:

Purpose

Provides top-level information about the floating-point implementation.
This register largely duplicates information held in the MIDR. Arm deprecates use of it.

Configuration

Implemented only if the implementation includes the Advanced SIMD and floating-point functionality.

Attributes

FPSID is a 32-bit register.

Field descriptions

31	24	23	22	16	15	8	7	4	3	0
Implementer		SW	Subarchitecture			PartNum		Variant		Revision

Implementer, bits [31:24]

Implementer codes are the same as those used for the MIDR.
For an implementation by Arm this field is 0x41, the ASCII code for A.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

SW, bit [23]

Software bit.
The value of this field is an IMPLEMENTATION DEFINED choice of:

SW	Meaning
0b0	The implementation provides a hardware implementation of the floating-point instructions.
0b1	The implementation supports only software emulation of the floating-point instructions.

In Armv8, the only permitted value is 0.
Access to this field is RO.

Subarchitecture, bits [22:16]

Subarchitecture version number.
The value of this field is an IMPLEMENTATION DEFINED choice of:

Subarchitecture	Meaning
0b00000000	VFPv1 architecture with an IMPLEMENTATION DEFINED subarchitecture.
0b00000001	VFPv2 architecture with Common VFP subarchitecture v1.
0b00000010	VFPv3 architecture, or later, with Common VFP subarchitecture v2. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.
0b00000011	VFPv3 architecture, or later, with Null subarchitecture. The entire floating-point implementation is in hardware, and no software support code is required. The VFP architecture version is indicated by the MVFR0 and MVFR1 registers. This value can be used only by an implementation that does not support the trap enable bits in the FPSCR .
0b00000100	VFPv3 architecture, or later, with Common VFP subarchitecture v3, and support for trap enable bits in FPSCR . The VFP architecture version is indicated by the MVFR0 and MVFR1 registers.

For a subarchitecture designed by Arm the most significant bit of this field, register bit[22], is 0. Values with a most significant bit of 0 that are not listed here are reserved.

When the subarchitecture designer is not Arm, the most significant bit of this field, register bit[22], must be 1. Each implementer must maintain its own list of subarchitectures it has designed, starting at subarchitecture version number 0x40.

In Armv8, the permitted values are 0b00000011 and 0b00000100.

Access to this field is RO.

PartNum, bits [15:8]

Part Number for the floating-point implementation, assigned by the implementer.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [7:4]

Variant number. Typically, this field distinguishes between different production variants of a single product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the floating-point implementation.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing FPSID

Accesses to this register use the following encodings in the System register encoding space:

VMRS {<c>} {<q>} <Rt>, <spec_reg>

reg

0b0000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    elsif EL2Enabled() && HCR().TID0 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        R(t) = FPSID();
    end;
elsif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        R(t) = FPSID();
    end;
end;
```

VMSR{<c>}{<q>} <spec_reg>, <Rt>

reg

0b0000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    elsif EL2Enabled() && HCR().TID0 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        return;
    end;
elsif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        return;
    end;
end;
```

D10.2.49 HACR, Hyp Auxiliary Configuration Register

The HACR characteristics are:

Purpose

Controls trapping to Hyp mode of IMPLEMENTATION DEFINED aspects of Non-secure EL1 or EL0 operation.

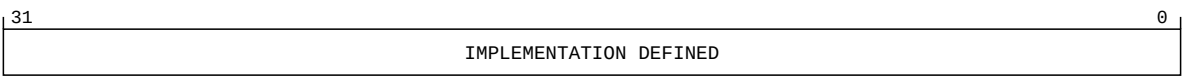
Configuration

There are no configuration notes.

Attributes

HACR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b111

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HACR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HACR() = R(t);
end;
```


D10.2.50 HACTLR, Hyp Auxiliary Control Register

The HACTLR characteristics are:

Purpose

Controls IMPLEMENTATION DEFINED features of Hyp mode operation.

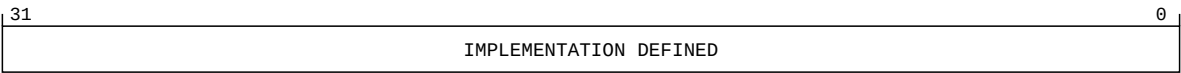
Configuration

There are no configuration notes.

Attributes

HACTLR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACTLR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HACTLR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HACTLR() = R(t);
end;
```

D10.2.51 HACTLR2, Hyp Auxiliary Control Register 2

The HACTLR2 characteristics are:

Purpose

Provides additional space to the HACTLR register to hold IMPLEMENTATION DEFINED trap functionality.

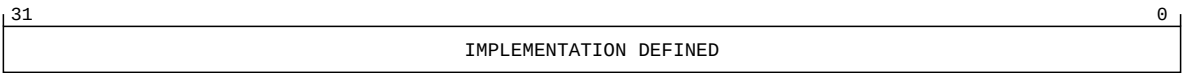
Configuration

In Armv8.0, it is IMPLEMENTATION DEFINED whether this register is implemented, or whether it causes UNDEFINED exceptions when accessed. The implementation of this register can be detected by examining [ID_MMFR4.AC2](#).

Attributes

HACTLR2 is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACTLR2

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HACTLR2();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HACTLR2() = R(t);
end;
```

D10.2.52 HADFSR, Hyp Auxiliary Data Fault Status Register

The HADFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED syndrome information for Data Abort exceptions taken to Hyp mode.

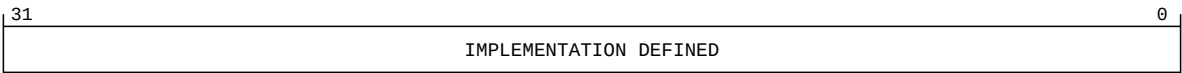
Configuration

This is an optional register. An implementation that does not require this register can implement it as RES0.

Attributes

HADFSR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HADFSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HADFSR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HADFSR() = R(t);
end;
```

D10.2.53 HAIFSR, Hyp Auxiliary Instruction Fault Status Register

The HAIFSR characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED syndrome information for Prefetch Abort exceptions taken to Hyp mode.

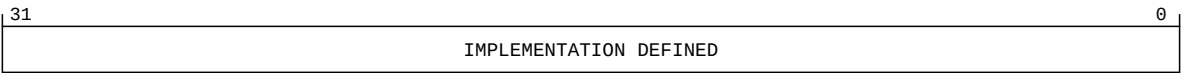
Configuration

This is an optional register. An implementation that does not require this register can implement it as RES0.

Attributes

HAIFSR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HAIFSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HAIFSR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HAIFSR() = R(t);
end;
```


D10.2.54 HMAIR0, Hyp Auxiliary Memory Attribute Indirection Register 0

The HMAIR0 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by [HMAIR0](#). These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in [HMAIR0](#).

Configuration

There are no configuration notes.

Attributes

HMAIR0 is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HMAIR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HMAIR0();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HAMAIR0() = R(t);
end;
```

D10.2.55 HMAIR1, Hyp Auxiliary Memory Attribute Indirection Register 1

The HMAIR1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory attribute encodings defined by [HMAIR1](#). These IMPLEMENTATION DEFINED attributes can only provide additional qualifiers for the memory attribute encodings, and cannot change the memory attributes defined in [HMAIR1](#).

Configuration

There are no configuration notes.

Attributes

HMAIR1 is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

If an implementation does not provide any IMPLEMENTATION DEFINED memory attributes, this register is RES0.

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HMAIR1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HMAIR1();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0011	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HAMAIR1() = R(t);
end;
```

D10.2.56 HCPTR, Hyp Architectural Feature Trap Register

The HCPTR characteristics are:

Purpose

Controls:

- Trapping to Hyp mode at EL1 or EL0, to trace, and to Advanced SIMD and floating-point functionality.
- Hyp mode access to trace, and to Advanced SIMD and floating-point functionality.

Note

Accesses to this functionality:

- From modes other than Hyp mode are also affected by settings in the CPACR and NSACR.
- From Hyp mode are also affected by settings in the NSACR.

Exceptions generated by the CPACR and NSACR controls are higher priority than those generated by the HCPTR controls.

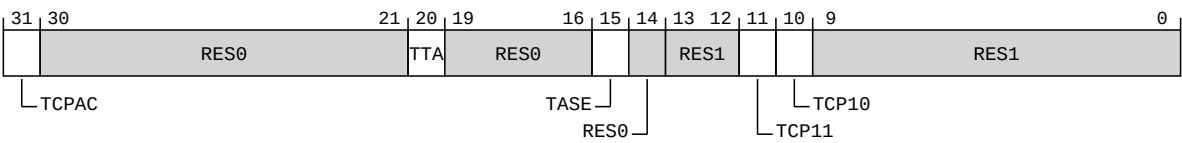
Configuration

There are no configuration notes.

Attributes

HCPTR is a 32-bit register.

Field descriptions



TCPAC, bit [31]

Traps EL1 MRC and MCR accesses to the CPACR to Hyp mode, reported using EC syndrome value 0x03.

TCPAC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the CPACR are trapped to Hyp mode.

Note

The CPACR is not accessible at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [30:21]

Reserved, RES0.

TTA, bit [20]

Traps System register MRC, MCR, MRRC, and MCRR accesses to all implemented trace System registers to Hyp mode, reported using EC syndrome values 0x05 and 0x0C.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any System register access to an implemented trace System register is trapped to Hyp mode, unless the access is trapped to EL1 by a CPACR or NSACR control, or the access is from EL0 and the definition of the register in the appropriate trace architecture specification indicates that the register is not accessible from EL0. A trapped instruction generates: - A Hyp Trap exception, if the exception is taken from EL0 or EL1. - An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

If the implementation does not include a trace unit, or does not include a System register interface to the trace unit registers, it is IMPLEMENTATION DEFINED whether this bit:

- Is RES0.
- Is RES1.
- Can be written from Hyp mode, and from Secure Monitor mode when SCR.NS is 1.

Note

- The ETMv4 architecture does not permit EL0 to access the trace System registers. If the trace unit implements FEAT_ETMv4, EL0 accesses to the trace System registers are UNDEFINED, and a resulting Undefined Instruction exception is higher priority than a HCPTR.TTA Hyp Trap exception.
- The architecture does not provide traps on trace register accesses through the optional memory-mapped debug interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [19:16]

Reserved, RES0.

TASE, bit [15]

Traps execution of Advanced SIMD instructions to Hyp mode, reported using EC syndrome value 0x07, when the value of HCPTR.TCP10 is 0.

TASE	Meaning
0b0	This control does not cause any instructions to be trapped.

TASE	Meaning
0b1	When the value of HCPTR.TCP10 is 0, any attempt to execute an Advanced SIMD instruction is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates: • A Hyp Trap exception, if the exception is taken from EL0 or EL1. • An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

When the value of HCPTR.TCP10 is 1, the value of this field is ignored.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

If the implementation includes floating-point functionality but does not include Advanced SIMD, this field is RES1.

Otherwise, it is IMPLEMENTATION DEFINED whether this field is implemented as a RW field. If it is not implemented as a RW field, then it is RAZ/WI.

For the list of instructions affected by this field, see ‘[Controls of Advanced SIMD operation that do not apply to floating-point operation](#)’.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [14]

Reserved, RES0.

Bits [13:12]

Reserved, RES1.

TCP11, bit [11]

When FEAT_FP is implemented and FEAT_AdvSIMD is implemented:

The value of this field is ignored. If this field is programmed with a different value to the TCP10 bit then this field is UNKNOWN on a direct read of the HCPTR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES1.

TCP10, bit [10]

When FEAT_FP is implemented and FEAT_AdvSIMD is implemented:

Trap accesses to Advanced SIMD and floating-point functionality to Hyp mode, reported using EC syndrome value 0x07:

TCP10	Meaning
0b0	This control does not cause any instructions to be trapped.

TCP10	Meaning
0b1	Any attempted access to Advanced SIMD and floating-point functionality is trapped to Hyp mode, unless it is trapped to EL1 by a CPACR or NSACR control. A trapped instruction generates: • A Hyp Trap exception, if the exception is taken from EL0 or EL1. • An Undefined Instruction exception taken to Hyp mode, if the exception is taken from Hyp mode.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the [FPSCR](#), [FPSID](#), [MVFR0](#), [MVFR1](#), [MVFR2](#), or [FPEXC](#) System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES1.

Bits [9:0]

Reserved, RES1.

Accessing HCPTR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b010

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HCPTR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HCPTR() = R(t);
end;
```

D10.2.57 HCR, Hyp Configuration Register

The HCR characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various operations are trapped to Hyp mode.

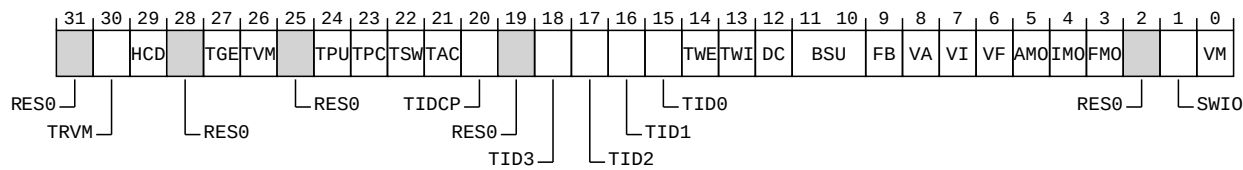
Configuration

There are no configuration notes.

Attributes

HCR is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES0.

TRVM, bit [30]

Trap Reads of Memory controls. Traps EL1 reads of the memory control registers to EL2.

MRC reads of the following registers are trapped and reported using EC syndrome value 0x03 and MRRC reads are trapped and reported using EC syndrome value 0x04:

[SCTLR](#), [DFSR](#), [IFSR](#), [DFAR](#), [IFAR](#), [ADFSR](#), [AIFSR](#), [MAIR0](#), [MAIR1](#), [AMAIR0](#), [AMAIR1](#), [CONTEXTIDR](#), [PRSELR](#), [PRBAR](#), [PRBAR<n>](#), [PRLAR](#), [PRLAR<n>](#).

TRVM	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 read accesses to the specified Memory control registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

HCD, bit [29]

HVC instruction disable.

Disables EL1 and EL2 execution of HVC instructions, reported using EC syndrome value 0x00.

HCD	Meaning
0b0	HVC instruction execution is enabled at EL2 and EL1.
0b1	HVC instructions are UNDEFINED at EL2 and EL1. The Undefined Instruction exception is taken to the Exception level at which the HVC instruction is executed.

Note
HVC instructions are always UNDEFINED at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [28]

Reserved, RES0.

TGE, bit [27]

Trap General Exceptions, from EL0.

TGE	Meaning
0b0	This control has no effect on execution at EL0.
0b1	Following statements are true: - All exceptions that would be routed to EL1 are routed to EL2. - The SCTLR.M bit is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR. - The HCR.{FMO, IMO, AMO} bits are treated as being 1 for all purposes other than returning the result of a direct read of HCR. - All virtual interrupts are disabled. - Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled. - An exception return to EL1 is treated as an illegal exception return.

Also, when HCR.TGE is 1:

- The [HDCR](#).{TDRA, TDOSA, TDA, TDE} bits are ignored and treated as being 1 other than for the purpose of a direct read of [HDCR](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TVM, bit [26]

Trap Memory controls. Traps EL1 writes to the memory control registers to EL2.

MCR writes of the following registers are trapped and reported using EC syndrome value 0x03 and MCRR writes are trapped and reported using EC syndrome value 0x04:

[SCTLR](#), [DFSR](#), [IFSR](#), [DFAR](#), [IFAR](#), [ADFSR](#), [AIFSR](#), [MAIR0](#), [MAIR1](#), [AMAIR0](#), [AMAIR1](#), [CONTEXTIDR](#), [PRSEL](#), [PRBAR](#), [PRBAR<n>](#), [PRLAR](#), [PRLAR<n>](#).

TVM	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 write accesses to the specified memory control registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [25]

Reserved, RES0.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps EL1 execution of those cache maintenance instructions to EL2.

MRC and MCR accesses of the following system instructions are trapped and reported using EC syndrome value 0x03:

- [ICIMVAU](#), [ICIALLU](#), [ICIALLUIS](#), [DCCMVAU](#).

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

TPU	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate to the Point of Unification instruction can be trapped when the value of this control is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TPC, bit [23]

Trap data or unified cache maintenance instructions that operate to the Point of Coherency. Traps EL1 execution of those cache maintenance instructions to EL2.

MRC and MCR accesses of the following system instructions are trapped and reported using EC syndrome value 0x03:

- [DCIMVAC](#), [DCCIMVAC](#), [DCCMVAC](#).

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

TPC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 execution of the specified cache maintenance instructions is trapped to EL2.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, invalidate, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TSW, bit [22]

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps EL1 execution of those cache maintenance instructions by set/way to EL2.

MRC and MCR accesses of the following system instructions are trapped and reported using EC syndrome value 0x03:

- [DCISW](#), [DCCSW](#), [DCCISW](#).

Note

An Undefined Instruction exception generated at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

TSW	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 execution of the specified cache maintenance instructions is trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TAC, bit [21]

Trap Auxiliary Control Registers. Traps EL1 accesses to the Auxiliary Control Registers to EL2.

MRC and MCR accesses of the following registers are trapped and reported using EC syndrome value 0x03:

[ACTLR](#) and, if implemented, [ACTLR2](#).

TAC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the specified registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps EL1 accesses to the encodings for IMPLEMENTATION DEFINED System Registers to EL2.

MRC and MCR accesses of the following encodings are trapped and reported using EC syndrome value 0x03:

- All coproc==p15, CRn==c9, Opcode1 = {0-7}, CRm == {c0-c2, c5-c8}, opcode2 == {0-7}.
- All coproc==p15, CRn==c10, Opcode1 == {0-7}, CRm == {c0, c1, c4, c8}, opcode2 == {0-7}.
- All coproc==p15, CRn==c11, Opcode1=={0-7}, CRm == {c0-c8, c15}, opcode2 == {0-7}.

TIDCP	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the specified System register encodings for IMPLEMENTATION DEFINED functionality are trapped to EL2.

When HCR.TIDCP is set to 1, it is IMPLEMENTATION DEFINED whether any of this functionality accessed from EL0 is trapped to EL2. Otherwise, it is UNDEFINED and the PE takes an Undefined Instruction exception to Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [19]

Reserved, RES0.

TID3, bit [18]

Trap ID group 3. Traps EL1 reads of the following registers to EL2 as follows:

- VMRS access to [MVFR0](#), [MVFR1](#), and [MVFR2](#), reported using EC syndrome value 0x08, unless access is also trapped by [HCPTR](#) which takes priority.
- MRC access to the following registers are reported using EC syndrome value 0x03:
 - [ID_PFR0](#), [ID_PFR1](#), [ID_DFR0](#), [ID_AFR0](#), [ID_MMFR0](#), [ID_MMFR1](#), [ID_MMFR2](#), [ID_MMFR3](#), [ID_ISAR0](#), [ID_ISAR1](#), [ID_ISAR2](#), [ID_ISAR3](#), [ID_ISAR4](#), and [ID_ISAR5](#).
 - [ID_MMFR4](#) is trapped to EL2, unless implemented as RAZ, when it is IMPLEMENTATION DEFINED whether accesses to [ID_MMFR4](#) is trapped.
- Otherwise, it is IMPLEMENTATION DEFINED whether this bit traps MRC accesses to registers not already mentioned, with coproc == p15, opc1 == 0, CRn == c0, CRm == {c2-c7}, opc2 == {0-7}, reported with EC syndrome value 0x03.

TID3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified EL1 read accesses to ID group 3 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TID2, bit [17]

Trap ID group 2. Traps the following register MRC and MCR accesses to EL2, reported using EC syndrome value 0x03:

- EL1 and EL0 reads of the [CTR](#), [CCSIDR](#), [CLIDR](#), and [CSSELR](#).
- EL1 and EL0 writes to the [CSSELR](#).

TID2	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified EL1 and EL0 accesses to ID group 2 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TID1, bit [16]

Trap ID group 1. Traps EL1 MRC reads of the following registers to EL2, reported using EC syndrome value 0x03:

[TCMTR](#), [TLBTR](#), [REVIDR](#), [AIDR](#), [MPUIR](#).

TID1	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified EL1 read accesses to ID group 1 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TID0, bit [15]

Trap ID group 0. Traps the following register accesses to EL2:

- EL1 VMRS reads of [FPSID](#) reported using EC syndrome value 0x08.
- EL0 and EL1 MCR and MRC accesses of [JIDR](#) reported using EC syndrome value 0x05.

Note

- It is IMPLEMENTATION DEFINED whether the [JIDR](#) is RAZ or UNDEFINED at EL0. If it is UNDEFINED at EL0 then the Undefined Instruction exception takes precedence over this trap.
- The [FPSID](#) is not accessible at EL0.
- Writes to the [FPSID](#) are ignored, and not trapped by this control.

TID0	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified EL1 read accesses to ID group 0 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TWE, bit [14]

Traps EL0 and EL1 execution of WFE instructions to EL2, reported using EC syndrome value 0x01.

TWE	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFE instruction at EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE .

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TWI, bit [13]

Traps EL0 and EL1 execution of WFI instructions to EL2, reported using EC syndrome value 0x01.

TWI	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFI instruction at EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI .

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

DC, bit [12]

Default Cacheability.

DC	Meaning
0b0	This control has no effect on the EL1&0 translation regime.
0b1	This control has following effects: • The HCR.VM field behaves as 1 for all purposes other than a direct read of the value of the field. • The memory type produced by the first stage of the EL1&0 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate. • For instruction accesses, XN behaves as 0, meaning execution is permitted.

This field has no effect on the EL2 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from EL1 or EL0:

BSU	Meaning
0b00	No effect.
0b01	Inner Shareable.
0b10	Outer Shareable.
0b11	Full system.

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

FB, bit [9]

Force broadcast. Causes the following instructions to be broadcast within the Inner Shareable domain when executed from EL1:

[BPIALL](#), [ICIALLU](#).

FB	Meaning
0b0	This field has no effect on the operation of the specified instructions.
0b1	When one of the specified instruction is executed at EL1, the instruction is broadcast within the Inner Shareable shareability domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

VA, bit [8]

Virtual SError exception.

VA	Meaning
0b0	This mechanism is not making a virtual SError exception pending.
0b1	A virtual SError exception is pending because of this mechanism.

The virtual SError exception is enabled only when the value of HCR.{TGE, AMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

VI, bit [7]

Virtual IRQ exception.

VI	Meaning
0b0	This mechanism is not making a virtual IRQ pending.
0b1	A virtual IRQ is pending because of this mechanism.

The virtual IRQ is enabled only when the value of HCR.{TGE, IMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

VF, bit [6]

Virtual FIQ exception.

VF	Meaning
0b0	This mechanism is not making a virtual FIQ pending.

VF	Meaning
0b1	A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR.{TGE, FMO} is {0, 1}.

The Guest OS cannot distinguish the virtual exception from the corresponding physical exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

AMO, bit [5]

SError exception Mask Override. When this bit is set to 1, it overrides the effect of CPSR.A, and enables virtual exception signaling by the VA bit.

If the value of HCR.TGE is 0, then virtual SError exceptions are enabled.

If the value of HCR.TGE is 1, then the HCR.AMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

IMO, bit [4]

IRQ Mask Override. When this bit is set to 1, it overrides the effect of CPSR.I, and enables virtual exception signaling by the VI bit.

If the value of HCR.TGE is 0, then Virtual IRQ interrupts are enabled.

If the value of HCR.TGE is 1, then HCR.IMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

FMO, bit [3]

FIQ Mask Override. When this bit is set to 1, it overrides the effect of CPSR.F, and enables virtual exception signaling by the VF bit.

If the value of HCR.TGE is 0, then Virtual FIQ interrupts are enabled.

If the value of HCR.TGE is 1, then HCR.FMO bit behaves as 1 for all purposes other than a direct read of the value of the bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [2]

Reserved, RES0.

SWIO, bit [1]

Set/Way Invalidation Override. Causes EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way.

SWIO	Meaning
0b0	This control has no effect on the operation of data cache invalidate by set/way instructions.
0b1	Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When this bit is set to 1, [DCISW](#) performs the same invalidation as a [DCCISW](#) instruction.

As a result of changes to the behavior of [DCISW](#), this bit is redundant in Armv8. This bit can be implemented as RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

VM, bit [0]

Virtualization enable. Enables stage 2 address protection for EL1&0 translation regime.

VM	Meaning
0b0	EL1&0 stage 2 address protection disabled.
0b1	EL1&0 stage 2 address protection enabled.

If the HCR.DC bit is set to 1, then the behavior of the PE when executing in other than Hyp mode is consistent with HCR.VM being 1, regardless of the actual value of HCR.VM, other than the value returned by an explicit read of HCR.VM.

When the value of this bit is 1, data cache invalidate instructions executed at EL1 perform a data cache clean and invalidate. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR.SWIO bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing HCR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HCR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HCR() = R(t);
end;
```

D10.2.58 HCR2, Hyp Configuration Register 2

The HCR2 characteristics are:

Purpose

Provides additional configuration controls for virtualization.

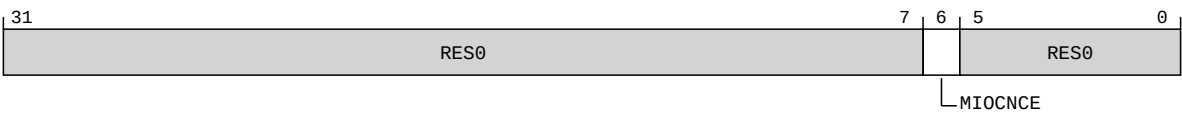
Configuration

There are no configuration notes.

Attributes

HCR2 is a 32-bit register.

Field descriptions



Bits [31:7]

Reserved, RES0.

MIOCNCNCE, bit [6]

Mismatched Inner/Outer Cacheable Non-Coherency Enable, for the EL1&0 translation regime.

MIOCNCNCE	Meaning
0b0	For the EL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there must be no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.
0b1	For the EL1&0 translation regime, for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there might be a loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

The value of this field has no effect on translation regimes other than the EL1&0 translation regime. This field can be implemented as RAZ/WI.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [5:0]

Reserved, RES0.

Accessing HCR2

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HCR2();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HCR2() = R(t);
end;
```

D10.2.59 HDFAR, Hyp Data Fault Address Register

The HDFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Data Abort exception that is taken to Hyp mode.

Configuration

There are no configuration notes.

Attributes

HDFAR is a 32-bit register.

Field descriptions

31	0
VA of faulting address of synchronous Data Abort exception taken to Hyp mode	

VA, bits [31:0]

VA of faulting address of synchronous Data Abort exception taken to Hyp mode.

On a Prefetch Abort exception, this register is UNKNOWN.

Any execution in a Non-secure EL1 or Non-secure EL0 mode makes this register UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HDFAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HDFAR();
end;
```


MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HDFAR() = R(t);
end;
```

D10.2.60 HIFAR, Hyp Instruction Fault Address Register

The HIFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception that is taken to Hyp mode.

Configuration

There are no configuration notes.

Attributes

HIFAR is a 32-bit register.

Field descriptions

31	0
VA of faulting address of synchronous Prefetch Abort exception taken to Hyp mode	

VA, bits [31:0]

VA of faulting address of synchronous Prefetch Abort exception taken to Hyp mode.

On a Data Abort exception, this register is UNKNOWN.

Any execution in a Non-secure EL1 or Non-secure EL0 mode makes this register UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HIFAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HIFAR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HIFAR() = R(t);
end;
```

D10.2.61 HMAIR0, Hyp Memory Attribute Indirection Register 0

The HMAIR0 characteristics are:

Purpose

Along with [HMAIR1](#), provides the memory attribute encodings corresponding to the possible AttrIdx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

AttrIdx[2] indicates the HMAIR register to be used:

- When AttrIdx[2] is 0, HMAIR0 is used.
- When AttrIdx[2] is 1, [HMAIR1](#) is used.

Configuration

There are no configuration notes.

Attributes

HMAIR0 is a 32-bit register.

Field descriptions

TTBCR.EAE=1

31	24	23	16	15	8	7	0
Attr3				Attr2			
Attr1				Attr0			

Attr<n>, bits [8n+7:8n], for n = 3 to 0

The memory attribute encoding for an AttrIdx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIdx[2:0] gives the value of <n> in Attr<n>.
- AttrIdx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
0b0000	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
0b00RW, RW not 0b00	Normal memory, Outer Write-Through Transient.
0b0100	Normal memory, Outer Non-cacheable.
0b01RW, RW not 0b00	Normal memory, Outer Write-Back Transient.
0b10RW	Normal memory, Outer Write-Through Non-transient.
0b11RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0b0000	Meaning when Attr<n>[7:4] is not 0b0000
0b0000	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
0b0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
0b1000	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=0b00)
0b10RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
0b1100	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=0b00)
0b11RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HMAIR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HMAIR0();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HMAIR0() = R(t);
end;
```

D10.2.62 HMAIR1, Hyp Memory Attribute Indirection Register 1

The HMAIR1 characteristics are:

Purpose

Along with [HMAIR0](#), provides the memory attribute encodings corresponding to the possible AttrIdx values in a Long-descriptor format translation table entry for stage 1 translations for memory accesses from Hyp mode.

AttrIdx[2] indicates the HMAIR register to be used:

- When AttrIdx[2] is 0, [HMAIR0](#) is used.
- When AttrIdx[2] is 1, HMAIR1 is used.

Configuration

There are no configuration notes.

Attributes

HMAIR1 is a 32-bit register.

Field descriptions

TTBCR.EAE=1

31	24	23	16	15	8	7	0
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			
Attr7				Attr6			

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0b0000	Meaning when Attr<n>[7:4] is not 0b0000
0b0000	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
0b0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
0b1000	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=0b00)
0b10RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
0b1100	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=0b00)
0b11RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HMAIR1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HMAIR1();
end;
```


MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HMAIR1() = R(t);
end;
```

D10.2.63 HMPUIR, Hypervisor MPU Type Register

The HMPUIR characteristics are:

Purpose

Identifies the number of regions supported by the EL2 MPU.

Configuration

There are no configuration notes.

Attributes

HMPUIR is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REGION, bits [7:0]

The number of EL2 MPU regions supported.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing HMPUIR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HMPUIR();
end;
```

D10.2.64 HPFAR, Hyp IPA Fault Address Register

The HPFAR characteristics are:

Purpose

Holds the faulting IPA for some aborts on a stage 2 translation taken to Hyp mode.

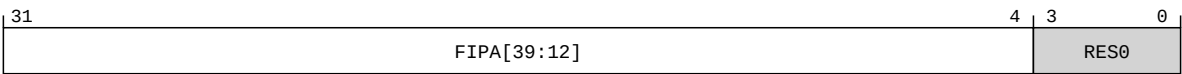
Configuration

There are no configuration notes.

Attributes

HPFAR is a 32-bit register.

Field descriptions



Execution in any Non-secure mode other than Hyp mode makes this register UNKNOWN.

FIPA[39:12], bits [31:4]

Bits [39:12] of the faulting intermediate physical address.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:0]

Reserved, RES0.

Accessing HPFAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = HPFAR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0000	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HPFAR() = R(t);
end;
```

D10.2.65 HPRBAR, Hypervisor Protection Region Base Address Register

The HPRBAR characteristics are:

Purpose

Provides access to the base addresses for the EL2 MPU region. [HPRSEL](#).REGION determines which MPU region is selected.

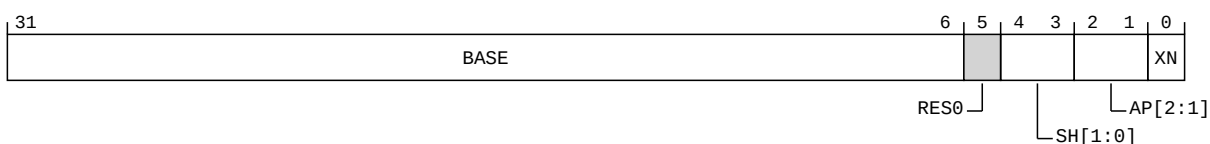
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

HPRBAR is a 32-bit register.

Field descriptions

**BASE, bits [31:6]**

Bits[31:6] of the lower inclusive limit of the selected EL2 MPU memory region. This value is zero extended to provide the base address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SH[1:0], bits [4:3]

Shareability attribute SH[1:0].

SH[1:0]	Meaning
0b00	Non-shareable
0b01	Reserved, CONSTRAINED UNPREDICTABLE
0b10	Outer Shareable
0b11	Inner Shareable

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AP[2:1], bits [2:1]

Access Permission attributes AP[2:1].

AP[2:1]	Meaning
0b00	Read/write at EL2, no access at EL1 or EL0
0b01	Read/write at EL2, EL1 and EL0
0b10	Read-only at EL2, no access at EL1 or EL0
0b11	Read-only at EL2, EL1 and EL0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

XN, bit [0]

Execute Never

XN	Meaning
0b0	Execution of instructions fetched from the region is permitted.
0b1	Execution of instructions fetched from the region is not permitted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HPRBAR

Any access to MPU region register HPRBAR above the number of implemented regions specified by [HMPUIR.REGION](#) is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented HPRBAR register return an UNKNOWN value.
- Writes to unimplemented HPRBAR register make all HPRBAR registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0011	0b000

```
if UInt(HPRSELR().REGION) >= UInt(HMPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
```

```
        end;  
    elsif PSTATE.EL == EL2 then  
        R(t) = HPRBAR(UInt(HPRSELR().REGION));  
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0011	0b000

```
if UInt(HPRSELR().REGION) >= UInt(HMPUIR().REGION) then  
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURRegion_RW_UNKNOWN);  
elsif PSTATE.EL == EL0 then  
    Undefined();  
elsif PSTATE.EL == EL1 then  
    if EL2Enabled() && HSTR().T6 == '1' then  
        AArch32_TakeHypTrapException(0x03);  
    else  
        Undefined();  
    end;  
elsif PSTATE.EL == EL2 then  
    HPRBAR(UInt(HPRSELR().REGION)) = R(t);  
end;
```

D10.2.66 HPRBAR<n>, Hypervisor Protection Region Base Address Register n, n = 0 - 31

The HPRBAR<n> characteristics are:

Purpose

Provides access to the base address for the EL2 MPU region determined by the value of ‘n’.

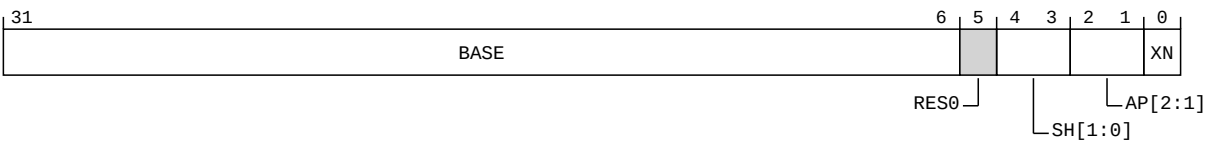
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

HPRBAR<n> is a 32-bit register.

Field descriptions



BASE, bits [31:6]

Bits[31:6] of the lower inclusive limit of the selected EL2 MPU memory region. This value is zero extended to provide the base address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SH[1:0], bits [4:3]

Shareability attribute SH[1:0].

SH[1:0]	Meaning
0b00	Non-shareable
0b01	Reserved, CONSTRAINED UNPREDICTABLE
0b10	Outer Shareable
0b11	Inner Shareable

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AP[2:1], bits [2:1]

Access Permission attributes AP[2:1].

AP[2:1]	Meaning
0b00	Read/write at EL2, no access at EL1 or EL0
0b01	Read/write at EL2, EL1 and EL0
0b10	Read-only at EL2, no access at EL1 or EL0
0b11	Read-only at EL2, EL1 and EL0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

XN, bit [0]

Execute Never

XN	Meaning
0b0	Execution of instructions fetched from the region is permitted.
0b1	Execution of instructions fetched from the region is not permitted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HPRBAR<n>

Any access to MPU region register HPRBAR<n> above the number of implemented regions specified by [HMPUIR.REGION](#) is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented HPRBAR<n> return an UNKNOWN value.
- Writes to unimplemented HPRBAR<n> register make all HPRBAR registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where n = 0-31

coproc	opc1	CRn	CRm	opc2
0b1111	0b10:n[4]	0b0110	0b1:n[3:1]	n[0]:0b00

```
let n:integer = UInt(opc1[0] :: CRm[2:0] :: opc2[2]);

if n >= UInt(HMPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
```

```
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = HPRBAR(n);
    end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where n = 0-31

coproc	opc1	CRn	CRm	opc2
0b1111	0b10:n[4]	0b0110	0b1:n[3:1]	n[0]:0b00

```
let n:integer = UInt(opc1[0] :: CRm[2:0] :: opc2[2]);

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HPRBAR(n) = R(t);
end;
```

D10.2.67 HPRENR, Hypervisor Protection Region Enable Register

The HPRENR characteristics are:

Purpose

Provides direct access to the [HPRLAR](#).EN bits of EL2 MPU regions from 0 to 31.

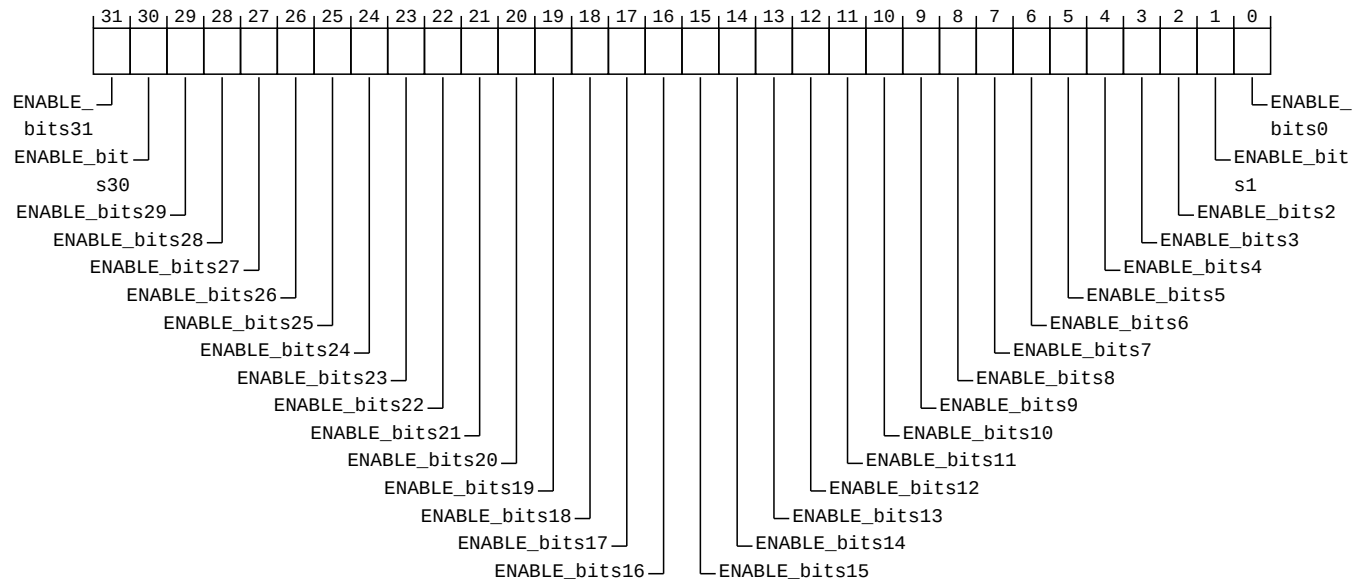
Configuration

There are no configuration notes.

Attributes

HPRENR is a 32-bit register.

Field descriptions



ENABLE_bits<n> , bits [n], for n = 31 to 0

Enable bit. Each bit, n, enables or disables the respective EL2 MPU region. The bits associated with the unimplemented MPU regions are RAZ/WI.

ENABLE_bits<n>	Meaning
0b0	Disables the EL2 MPU n region.
0b1	Enables the EL2 MPU n region.

The reset behavior of this field is:

- On a Warm reset, this field resets to the expression 0x00000000.

Accessing HPRENR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HPRENR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HPRENR() = R(t);
end;
```

D10.2.68 HPRLAR, Hypervisor Protection Region Limit Address Register

The HPRLAR characteristics are:

Purpose

Provides access to the limit addresses for the EL2 MPU region. [HPRSEL.R.REGION](#) determines which MPU region is selected.

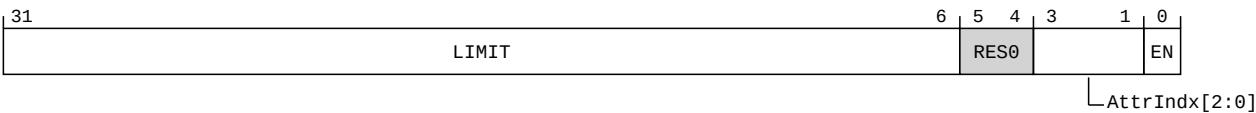
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

HPRLAR is a 32-bit register.

Field descriptions



LIMIT, bits [31:6]

Bits[31:6] of the upper inclusive limit of the selected EL2 MPU memory region. This value is concatenated with the value 0x3F to provide the limit address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [5:4]

Reserved, RES0.

AttrIndx[2:0], bits [3:1]

AttrIndx[2:0] selects attributes from within the associated Memory Attribute Indirection Register.

AttrIndx[2:0]	Meaning
0b000	Select the Attr0 field from HMAIR0.
0b001	Select the Attr1 field from HMAIR0.
0b010	Select the Attr2 field from HMAIR0.
0b011	Select the Attr3 field from HMAIR0.
0b100	Select the Attr4 field from HMAIR1.
0b101	Select the Attr5 field from HMAIR1.

AttrIdx[2:0]	Meaning
0b110	Select the Attr6 field from HMAIR1.
0b111	Select the Attr7 field from HMAIR1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

EN	Meaning
0b0	Region disabled.
0b1	Region enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing HPRLAR

Any access to MPU region register HPRLAR above the number of implemented regions specified by [HMPUIR](#).REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented HPRLAR register return an UNKNOWN value.
- Writes to unimplemented HPRLAR register make all HPRLAR registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0011	0b001

```

if UInt(HPRSEL().REGION) >= UInt(HMPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HPRLAR(UInt(HPRSEL().REGION));
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0011	0b001

```
if UInt(HPRSELR().REGION) >= UInt(HMPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURRegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HPRLAR(UInt(HPRSELR().REGION)) = R(t);
end;
```

D10.2.69 HPRLAR<n>, Hypervisor Protection Region Limit Address Register n, n = 0 - 31

The HPRLAR<n> characteristics are:

Purpose

Provides access to the limit address for the EL2 MPU region determined by the value of ‘n’.

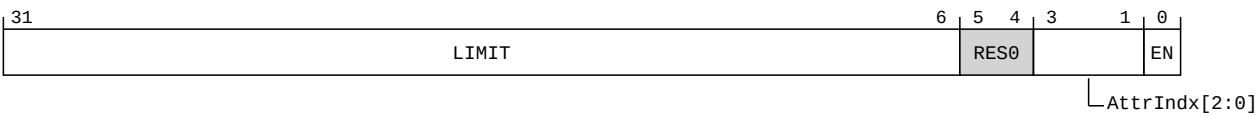
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

HPRLAR<n> is a 32-bit register.

Field descriptions



LIMIT, bits [31:6]

Bits[31:6] of the upper inclusive limit of the selected EL2 MPU memory region. This value is concatenated with the value 0x3F to provide the limit address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [5:4]

Reserved, RES0.

AttrIdx[2:0], bits [3:1]

AttrIdx[2:0] selects attributes from within the associated Memory Attribute Indirection Register.

AttrIdx[2:0]	Meaning
0b000	Select the Attr0 field from HMAIR0.
0b001	Select the Attr1 field from HMAIR0.
0b010	Select the Attr2 field from HMAIR0.
0b011	Select the Attr3 field from HMAIR0.
0b100	Select the Attr4 field from HMAIR1.
0b101	Select the Attr5 field from HMAIR1.

AttrIdx[2:0]	Meaning
0b110	Select the Attr6 field from HMAIR1.
0b111	Select the Attr7 field from HMAIR1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

EN	Meaning
0b0	Region disabled.
0b1	Region enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing HPRLAR<n>

Any access to MPU region register HPRLAR<n> above the number of implemented regions specified by [HMPUIR.REGION](#) is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented HPRLAR<n> return an UNKNOWN value.
- Writes to unimplemented HPRLAR<n> register make all HPRLAR registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where n = 0-31

coproc	opc1	CRn	CRm	opc2
0b1111	0b10:n[4]	0b0110	0b1:n[3:1]	n[0]:0b01

```
let n:integer = UInt(opc1[0] :: CRm[2:0] :: opc2[2]);

if n >= UInt(HMPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURRegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
```

```
        R(t) = HPRLAR(n);  
    end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where n = 0-31

coproc	opc1	CRn	CRm	opc2
0b1111	0b10:n[4]	0b0110	0b1:n[3:1]	n[0]:0b01

```
    let n:integer = UInt(opc1[0] :: CRm[2:0] :: opc2[2]);  
  
    if n >= UInt(HMPUIR().REGION) then  
        ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURRegion_RW_UNKNOWN);  
    elsif PSTATE.EL == EL0 then  
        Undefined();  
    elsif PSTATE.EL == EL1 then  
        if EL2Enabled() && HSTR().T10 == '1' then  
            AArch32_TakeHypTrapException(0x03);  
        else  
            Undefined();  
        end;  
    elsif PSTATE.EL == EL2 then  
        HPRLAR(n) = R(t);  
    end;
```

D10.2.70 HPRSELR, Hypervisor Protection Region Selection Register

The HPRSELR characteristics are:

Purpose

Selects the region number for the EL2 MPU region associated with the [HPRBAR](#) and [HPRLAR](#) registers.

Configuration

There are no configuration notes.

Attributes

HPRSELR is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REGION, bits [7:0]

The number of the current EL2 MPU region visible in [HPRBAR](#) and [HPRLAR](#). For N implemented MPU regions, memory region numbering starts at 0 and increments by 1 to the value N-1.

Writing a value greater than or equal to the number of implemented MPU regions specified by [HMPUIR.REGION](#), results in CONSTRAINED UNPREDICTABLE behavior.

CONSTRAINED UNPREDICTABLE behavior is that HPRSELR register becomes UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HPRSELR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
end;
```

```
elseif PSTATE.EL == EL2 then
    R(t) = HPRSELR();
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HPRSELR() = R(t);
end;
```

D10.2.71 HRMR, Hyp Reset Management Register

The HRMR characteristics are:

Purpose

A write to this register at EL2 can request a Warm reset.

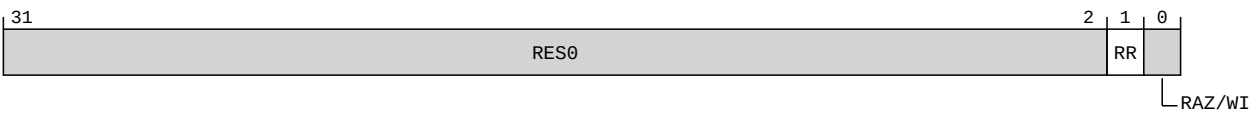
Configuration

It is IMPLEMENTATION DEFINED whether this register is implemented.

Attributes

HRMR is a 32-bit register.

Field descriptions



Bits [31:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [0]

Reserved, RAZ/WI.

Accessing HRMR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b010

```
if PSTATE.EL == EL1 && EL2Enabled() && HSTR().T12 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif PSTATE.EL == EL2 then
    R(t) = HRMR();
else
    Undefined();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b010

```
if PSTATE.EL == EL1 && EL2Enabled() && HSTR().T12 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif PSTATE.EL == EL2 then
    HRMR() = R(t);
else
    Undefined();
end;
```

D10.2.72 HSCTLR, Hyp System Control Register

The HSCTLR characteristics are:

Purpose

Provides top-level control of the system operation in Hyp mode.

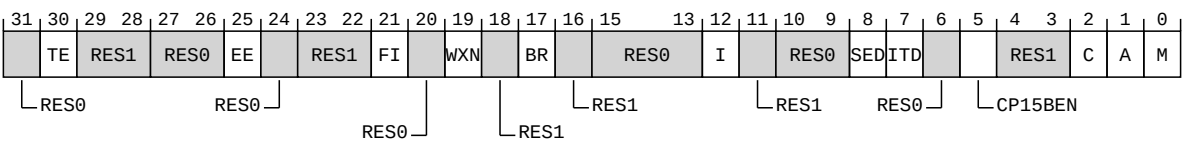
Configuration

There are no configuration notes.

Attributes

HSCTLR is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES0.

TE, bit [30]

T32 Exception Enable. This bit controls whether exceptions to EL2 are taken to A32 or T32 state:

TE	Meaning
0b0	Exceptions, including reset, taken to A32 state.
0b1	Exceptions, including reset, taken to T32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bits [29:28]

Reserved, RES1.

Bits [27:26]

Reserved, RES0.

EE, bit [25]

When FEAT_MixedEnd is implemented:

The value of the PSTATE.E bit on entry to Hyp mode.

EE	Meaning
0b0	Little-endian. PSTATE.E is cleared to 0 on entry to Hyp mode.
0b1	Big-endian. PSTATE.E is set to 1 on entry to Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_BigEnd is implemented:

Explicit data accesses at EL1 is big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL1 is little-endian.

Reserved, RES0.

Bit [24]

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

FI, bit [21]

Fast Interrupts enable.

FI	Meaning
0b0	All performance features enabled.
0b1	Low interrupt latency configuration. Some performance features disabled.

If HSCTLR.FI == 1, then asynchronous errors and interrupts can interrupt LDM/STM instructions that access Normal memory. The register that provides the base address is restored to its original value and any other modified registers become UNKNOWN.

Up to two MPUs can determine whether an access is to Normal memory, therefore the value of the FI field depends on whether the appropriate MPU or MPUs allow the access to the location as access to Normal memory.

— For EL0 and EL1 accesses, this is the combination of the attributes determined by the configuration settings of the EL1 MPU and the configuration settings of the EL2 MPU. — For EL2 accesses, these are the attributes determined by the configuration settings of the EL2 MPU.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 translation regime, this bit can force all memory regions that are writable to be treated as XN.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	Any region that is writable in the EL2 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when HSCTLR.M bit is set.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [18]

Reserved, RES1.

BR, bit [17]

Background Region enable for EL2.

BR	Meaning
0b0	EL2 MPU Background region disabled. Any EL2 transaction that does not match an EL2 MPU region results in a fault.
0b1	EL2 MPU Background region enabled. For EL2 transactions that do not match an EL2 MPU region, the EL2 Background region attributes are used.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [16]

Reserved, RES1.

Bits [15:13]

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2:

I	Meaning
0b0	All instruction access to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache. If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.

I	Meaning
0b1	All instruction access to Normal memory from EL2 can be cached at all levels of instruction and unified cache. If the value of HSCTLR.M is 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the PL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [11]

Reserved, RES1.

Bits [10:9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at EL2.

SED	Meaning
0b0	SETEND instruction execution is enabled at EL2.
0b1	SETEND instructions are UNDEFINED at EL2.

If the implementation does not support mixed-endian operation at EL2, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL2.

ITD	Meaning
0b0	All IT instruction functionality is enabled at EL2.

ITD	Meaning
0b1	Any attempt at EL2 to execute any of the following is UNDEFINED: - All encodings of the IT instruction with hw1[3:0]! = 1000. - All encodings of the subsequent instruction with the following values for hw1: 0b11xxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM. 0b1011xxxxxxxxxxxx: All instructions in ‘Miscellaneous 16-bit instructions’. 0b10100xxxxxxxxxxx: ADD Rd, PC, #imm 0b01001xxxxxxxxxxx: LDR Rd, [PC, #imm] 0b0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC. 0b010001xx1xxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn. These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block. It is IMPLEMENTATION DEFINED whether the IT instruction is treated as: - A 16-bit instruction, that can only be followed by another 16-bit instruction. - The first half of a 32-bit instruction. This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED. An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1, then behavior is CONSTRAINED UNPREDICTABLE. For more information, see ‘Changes to an ITD control by an instruction in an IT block’.

ITD is optional, but if it is implemented in the HSCTLR, then it must also be implemented in SCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement ITD, access to this field is RAZ/WI.

Bit [6]

Reserved, RES0.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from EL2:

CP15BEN	Meaning
0b0	EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.
0b1	EL2 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the HSCTLR, then it must also be implemented in SCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement CP15BEN, access to this field is RAO/WI.

Bits [4:3]

Reserved, RES1.

C, bit [2]

Cacheability control, for data accesses at EL2:

C	Meaning
0b0	All data access to Normal memory from EL2 are Non-cacheable for all levels of data and unified cache.
0b1	All data access to Normal memory from EL2 can be cached at all levels of data and unified cache.

This bit has no effect on the PL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2:

A	Meaning
0b0	Alignment fault checking disabled when executing at EL2. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element or data elements being accessed.
0b1	Alignment fault checking enabled when executing at EL2. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element or data elements being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MPU enable for EL2 address translation. Possible values of this bit are:

M	Meaning
0b0	EL2 address translation disabled. See the HSCTLR.I field for the behavior of instruction accesses to Normal memory.
0b1	EL2 address translation enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing HSCTLR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HSCTLR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HSCTLR() = R(t);
end;

```

D10.2.73 HSR, Hyp Syndrome Register

The HSR characteristics are:

Purpose

Holds syndrome information for an exception taken to Hyp mode.

Configuration

There are no configuration notes.

Attributes

HSR is a 32-bit register.

Field descriptions

31	26	25	24	0
EC	IL	ISS		

Execution in any Non-secure PE mode other than Hyp mode makes this register UNKNOWN.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of HSR is UNKNOWN. The value written to HSR must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about. Possible values of this field are:

EC	Meaning	ISS
0b000000	Unknown reason.	ISS encoding for exceptions with an unknown reason
0b000001	Trapped WFI or WFE instruction execution. Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.	ISS encoding for Exception from a WFI or WFE instruction
0b000011	Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC value 0b000000.	ISS encoding for Exception from an MCR or MRC access
0b000100	Trapped MCRR or MRRC access with (coproc==0b1111) that is not reported using EC value 0b000000.	ISS encoding for Exception from an MCRR or MRRC access
0b000101	Trapped MCR or MRC access with (coproc==0b1110).	ISS encoding for Exception from an MCR or MRC access
0b000110	Trapped LDC or STC access. The only architected uses of these instructions are: - An STC to write data to memory from DBGDTRRXint. - An LDC to read data from memory to DBGDTRTXint.	ISS encoding for Exception from an LDC or STC instruction

EC	Meaning	ISS
0b000111	Access to Advanced SIMD or floating-point functionality trapped by a HCPTR. {TASE, TCP10} control. Excludes exceptions generated because Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.	ISS encoding for Exception from an access to SIMD or floating-point functionality, resulting from HCPTR
0b001000	Trapped VMRS access, from ID group trap, that is not reported using EC value 0b000111.	ISS encoding for Exception from an MCR or MRC access
0b001100	Trapped MRRC access with (coproc==0b1110).	ISS encoding for Exception from an MCRR or MRRC access
0b001110	Illegal exception return to AArch32 state.	ISS encoding for Exception from an Illegal state or PC alignment fault
0b010001	Exception on SVC instruction execution in AArch32 state routed to EL2.	ISS encoding for Exception from HVC or SVC instruction execution
0b010010	HVC instruction execution in AArch32 state, when HVC is not disabled.	ISS encoding for Exception from HVC or SVC instruction execution
0b100000	Prefetch Abort from a lower Exception level.	ISS encoding for Exception from a Prefetch Abort
0b100001	Prefetch Abort taken without a change in Exception level.	ISS encoding for Exception from a Prefetch Abort
0b100010	PC alignment fault exception.	ISS encoding for Exception from an Illegal state or PC alignment fault
0b100100	Data Abort exception from a lower Exception level.	ISS encoding for Exception from a Data Abort
0b100101	Data Abort exception taken without a change in Exception level.	ISS encoding for Exception from a Data Abort

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction length bit. Indicates the size of the instruction that has been trapped to Hyp mode. When this bit is valid, possible values of this bit are:

IL	Meaning
0b0	16-bit instruction trapped.
0b1	32-bit instruction trapped.

This field is RES1 and not valid for the following cases:

- When the EC value is 0b000000, indicating an exception with an unknown reason.

- Prefetch Aborts.
- Data Abort exceptions for which the HSR.ISS.ISV field is 0.
- When the EC value is 0b001110, indicating an Illegal state exception.

The IL field is not valid and is UNKNOWN on an exception from a PC alignment fault.

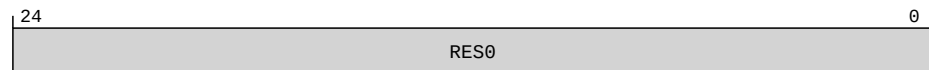
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

ISS encoding for exceptions with an unknown reason



Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for exceptions with an unknown reason

This EC value is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or is not accessible in the current PE mode in the current Security state, including:
 - A read access using a System register encoding pattern that is not allocated for reads or that does not permit reads in the current PE mode and Security state.
 - A write access using a System register encoding pattern that is not allocated for writes or that does not permit writes in the current PE mode and Security state.
 - Instruction encodings that are unallocated.
 - Instruction encodings for instructions not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- The attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the [SCTLR](#).{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
 - An HVC instruction when disabled by [HCR.HCD](#).
 - An HLT instruction when disabled by [EDSCR.HDE](#).
- An HVC instruction when disabled by [HCR.HCD](#). An HLT instruction when disabled by [EDSCR.HDE](#).
- An exception generated because of the attempted execution of an MSR (Banked register) or MRS (Banked register) instruction that would access a Banked register that is not accessible from the Security state and PE mode at which the instruction was executed.

Note

An exception is generated only if the CONSTRAINED UNPREDICTABLE behavior of the instruction is that it is UNDEFINED, see '[MSR \(banked register\) and MRS \(banked register\)](#)'.

- Attempted execution, in Debug state, of:
 - A DCPS1 instruction from EL0 when the value of [HCR.TGE](#) is 1.
 - A DCPS2 instruction at EL1 or EL0 when the value of [EDSCR.HDD](#) is 1.

‘Undefined Instruction exception, when the value of HCR.TGE is 1’ describes the configuration settings for a trap that returns an EC value of 0b0000000.

ISS encoding for Exception from a WFI or WFE instruction

24	23	20	19	1	0
CV	COND	RES0			TI

CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b11110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

TI	Meaning
0b0	WFI trapped.
0b1	WFE trapped.

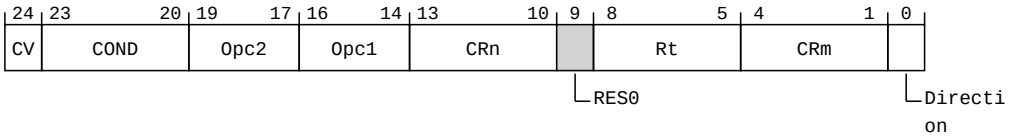
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from a WFI or WFE instruction

[HCR](#).{TWE, TWI} describe the configuration settings for this trap.

ISS encoding for Exception from an MCR or MRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

For a trapped VMRS access, holds the value 0b000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

For a trapped VMRS access, holds the value 0b111.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

For a trapped VMRS access, holds the reg field from the VMRS instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

Rt, bits [8:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

For a trapped VMRS access, holds the value 0b0000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write to System register space. MCR instruction.
0b1	Read from System register space. MRC or VMRS instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from an MCR or MRC access

The following fields describe configuration settings for traps from an MCR or MRC access using coproc 0b1111 that are reported using EC value 0b000011:

- [HCR](#).{TID1, TID2, TID3}, for Non-secure accesses to the ID registers at EL0 and EL1, trapped to EL2.
- [HCR](#).TIDCP, for Non-secure accesses to lockdown, DMA, and TCM operations at EL0 and EL1, trapped to EL2.
- [HCR](#).{TSW, TPC, TPU}, for Non-secure execution of cache maintenance instructions at EL1, trapped to EL2.
- [HCR](#).TLTB, for Non-secure execution of TLB maintenance instructions at EL1, trapped to EL2.
- [HCR](#).TAC, for Non-secure accesses to the Auxiliary Control Register at EL1, trapped to EL2.
- [HDCR](#).{TPM, TPMCR}, for Non-secure accesses to Performance Monitors System registers at EL0 and EL1, trapped to EL2.
- [HCPTR](#).TCPAC, for Non-secure accesses to the CPACR at EL1, trapped to EL2.
- [HCR](#).{TRVM, TVM}, for Non-secure accesses to virtual memory control registers at EL1, trapped to EL2.
- [HSTR](#).T<n>, for Non-secure accesses to System registers in the (coproc == 1111) encoding space at EL0 and EL1, trapped to EL2.
- [HDCR](#).TTRF, for Non-secure accesses to trace filter control registers from System register, trapped to EL2.
- [CNTHCTL](#).PL1PCEN, for Non-secure accesses to the Generic Timer registers at EL0 and EL1, trapped to EL2.

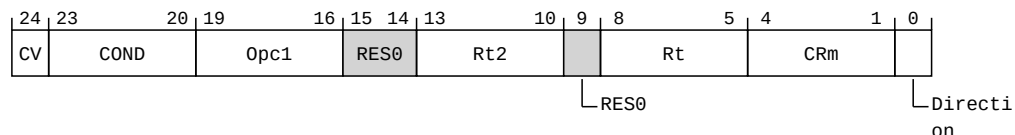
The following fields describe configuration settings for traps from an MCR or MRC access using coproc 0b1110 that are reported using EC value 0b000101:

- [HCR](#).TID0, for Non-secure accesses to the Primary device identification registers at EL0 and EL1, trapped to EL2.
- [HCPTR](#).TTA, for Non-secure accesses to trace System registers from System register, trapped to EL2.
- [HDCR](#).TDRA, for Non-secure accesses to Debug ROM registers from System register, trapped to EL2.
- [HDCR](#).TDOSA, for Non-secure accesses to powerdown debug System registers from System register, trapped to EL2.
- [HDCR](#).TDA, for Non-secure accesses to debug System registers from System register, trapped to EL2.

The following fields describes configuration settings for traps from a VMSR or VMRS access that are reported using EC value 0b001000:

- [HCR](#).TID0, for Non-secure accesses to the Primary device identification registers at EL1, for ID group traps trapped to EL2.
- [HCR](#).TID3, for Non-secure accesses to the Detailed feature identification registers at EL0 and EL1, for ID group traps trapped to EL2.
- [HCPTR](#).{TCP10, TCP11}, for Non-secure accesses to [FPSCR](#), [FPSID](#), [FPEXC](#), [MVFR0](#), [MVFR1](#), and [MVFR2](#), trapped to EL2.

ISS encoding for Exception from an MCRR or MRRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:14]

Reserved, RES0.

Rt2, bits [13:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

Rt, bits [8:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write to System register space. MCRR instruction.
0b1	Read from System register space. MRRC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from an MCRR or MRRC access

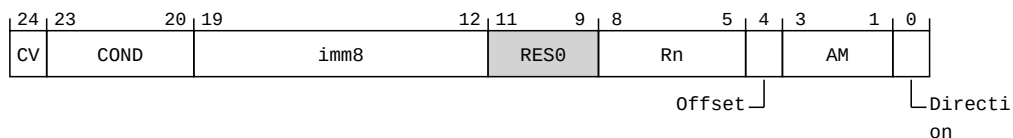
The following fields describe configuration settings for traps from an MCRR or MRRC access using coproc 0b1111 that are reported using EC value 0b000100:

- [HCR](#).{TRVM, TVM}, for Non-secure accesses to virtual memory control registers at EL1, trapped to EL2.
- [HDCR](#).TPM, for Non-secure accesses to Performance Monitors System registers at EL0 and EL1, trapped to EL2.
- [HCPTR](#).TAM, for Non-secure accesses to Activity Monitors System registers at EL0 and EL1, trapped to EL2.
- [CNTHCTL](#).{PL1PCEN, PL1PCTEN}, for Non-secure accesses to the Generic Timer registers at EL0 and EL1, trapped to EL2.
- [HSTR](#).T<n>, for Non-secure accesses to System registers in the (coproc == 1111) encoding space at EL0 and EL1, trapped to EL2.

The following fields describe configuration settings for traps from an MCRR or MRRC access using coproc 0b1110 that are reported using EC value 0b001100:

- [HCPTR](#).TTA, for Non-secure accesses to trace System registers from System register, trapped to EL2.
- [HDCR](#).TDRA, for Non-secure accesses to Debug ROM registers from System register, trapped to EL2.

ISS encoding for Exception from an LDC or STC instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:9]

Reserved, RES0.

Rn, bits [8:5]

The Rn value from the issued instruction. Valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction.

When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

Offset	Meaning
0b0	Subtract offset.
0b1	Add offset.

This bit corresponds to the U bit in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

AM	Meaning
0b000	Immediate unindexed.
0b001	Immediate post-indexed.
0b010	Immediate offset.
0b011	Immediate pre-indexed.
0b100	Literal unindexed. LDC instruction in A32 instruction set only. For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
0b110	Literal offset. LDC instruction only. For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

Bit [2] in this subfield indicates the instruction form, immediate or literal.

Bits [1:0] in this subfield correspond to the bits {P, W} in the instruction encoding.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write to memory. STC instruction.
0b1	Read from memory. LDC instruction.

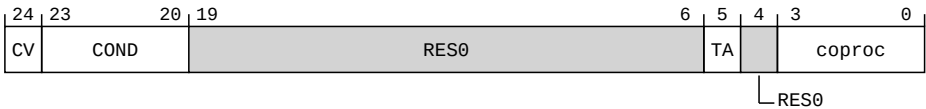
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from an LDC or STC instruction

[HDCR](#).TDA describes the configuration settings for the trap that is reported using EC value 0b000110.

ISS encoding for Exception from an access to SIMD or floating-point functionality, resulting from HCPTR



Excludes exceptions that occur because Advanced SIMD and floating-point functionality is not implemented, or because the value of [HCR](#).TGE is 1. These are reported with EC value 0b000000.

CV, bit [24]

Condition code valid. Possible values of this bit are:

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

When an A32 instruction is trapped, CV is set to 1.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether CV is set to 1 or set to 0. For more information, see the description of the COND field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction.

When an A32 instruction is trapped, CV is set to 1 and:

- If the instruction is conditional, COND is set to the condition code field value from the instruction.
- If the instruction is unconditional, COND is set to 0b1110.

A conditional A32 instruction that is known to pass its condition code check can be presented either:

- With COND set to 0b1110, the value for unconditional.
- With the COND value held in the instruction.

When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:

- CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
- CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.

For an implementation that, for both T32 and A32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:6]

Reserved, RES0.

TA, bit [5]

Indicates trapped use of Advanced SIMD functionality.

TA	Meaning
0b0	Exception was not caused by trapped use of Advanced SIMD functionality.
0b1	Exception was caused by trapped use of Advanced SIMD functionality.

Any use of an Advanced SIMD instruction that is not also a floating-point instruction that is trapped to Hyp mode because of a trap configured in the HCPTR sets this bit to 1.

For a list of these instructions, see ‘Controls of Advanced SIMD operation that do not apply to floating-point operation’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [4]

Reserved, RES0.

coproc, bits [3:0]

When the HSR.TA field returns the value 1, this field returns the value 0b1010. Otherwise, this field is RES0.

The reset behavior of this field is:

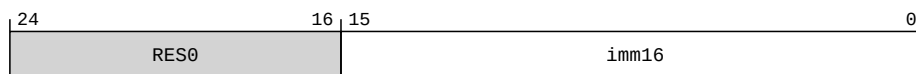
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from an access to SIMD or floating-point functionality, resulting from HCPTR

The following fields describe the configuration settings for the traps that are reported using EC value 0b000111:

- [HCPTR](#).{TCP11, TCP10}, for Non-secure accesses to the SIMD and floating-point registers, trapped to EL2.
- [HCPTR](#).TASE, for Non-secure accesses to Advanced SIMD functionality, trapped to EL2.

ISS encoding for Exception from HVC or SVC instruction execution



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, this is the value of the imm16 field of the issued instruction.

For an SVC instruction:

- If the instruction is unconditional, then:
 - For the T32 instruction, this field is zero-extended from the imm8 field of the instruction.
 - For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- For the T32 instruction, this field is zero-extended from the imm8 field of the instruction. For the A32 instruction, this field is the bottom 16 bits of the imm24 field of the instruction.
- If the instruction is conditional, this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from HVC or SVC instruction execution

The HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

‘[Supervisor Call exception, when the value of HCR.TGE is 1](#)’ describes the configuration settings for the trap reported with EC value 0b010001.

ISS encoding for Exception from a Prefetch Abort



Bits [24:11]

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort.

FnV	Meaning
0b0	HIFAR is valid.
0b1	HIFAR is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code. Possible values of this field are:

IFSC	Meaning
0b000100	Translation fault.
0b001100	Permission fault.
0b010000	Synchronous External abort, other than synchronous parity or ECC error.
0b011000	Synchronous parity or ECC error on memory access.
0b100010	Debug exception, only when the EC value is 0b100001.

All other values are reserved.

The effect of programming this field to a reserved value is that behavior is **CONSTRAINED UNPREDICTABLE**.

The reset behavior of this field is:

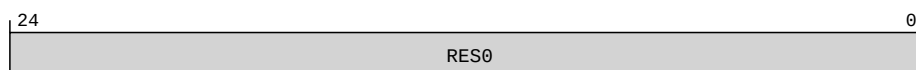
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from a Prefetch Abort

The following sections describe cases where Prefetch Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value 0b100000:

- ‘Abort exceptions, when the value of HCR.TGE is 1’.
- ‘Routing debug exceptions to EL2 using AArch32’.

ISS encoding for Exception from an Illegal state or PC alignment fault



Bits [24:0]

Reserved, RES0.

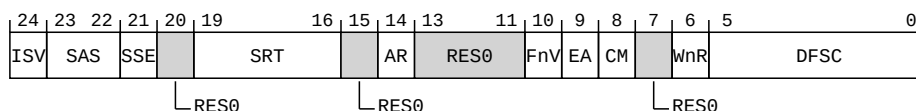
Additional Information for ISS encoding for Exception from an Illegal state or PC alignment fault

For more information about the Illegal state exception, see:

- ‘Illegal changes to PSTATE.M’.
- ‘Illegal return events from AArch32 state’.
- ‘Legal returns that set PSTATE.IL to 1’.
- ‘The Illegal Execution state exception’.

For more information about the PC alignment fault exception, see ‘Branching to an unaligned PC’.

ISS encoding for Exception from a Data Abort



ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

ISV	Meaning
0b0	No valid instruction syndrome. ISS[23:14] are RES0.
0b1	ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults except Data Abort exceptions generated by stage 2 address translations for which all the following apply to the instruction that generated the Data Abort exception:

- The instruction is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
- The instruction is not performing register writeback.
- The instruction is not using the PC as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in Memory access mode, as described in ‘Data Abort exceptions in Memory access mode’, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

Note

In the A32 instruction set, LDR*T and STR*T instructions always perform register writeback and therefore never return a valid instruction syndrome.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

Syndrome Access Size. When ISV is 1, indicates the size of the access attempted by the faulting operation.

SAS	Meaning
0b00	Byte
0b01	Halfword
0b10	Word
0b11	Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

SSE	Meaning
0b0	Sign-extension not required.
0b1	Data item must be sign-extended.

For all other operations this bit is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [20]

Reserved, RES0.

SRT, bits [19:16]

Syndrome Register Transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

AR	Meaning
0b0	Instruction did not have acquire/release semantics.
0b1	Instruction did have acquire/release semantics.

This field is UNKNOWN when the value of ISV is UNKNOWN.

This field is RES0 when the value of ISV is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [13:11]

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort.

FnV	Meaning
0b0	HDFAR is valid.
0b1	HDFAR is not valid, and holds an UNKNOWN value.

This field is valid only if DFSC is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External Abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache Maintenance. For a synchronous fault, identifies fault that comes from a cache maintenance or address translation instruction. For synchronous faults, the possible values of this bit are:

CM	Meaning
0b0	Fault not generated by a cache maintenance or address translation instruction.
0b1	Fault generated by a cache maintenance or address translation instruction.

For an asynchronous Data Abort exception, this bit is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by a write instruction or a read instruction.

WnR	Meaning
0b0	Abort caused by a read instruction.
0b1	Abort caused by a write instruction.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For an asynchronous Data Abort exception this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code. Possible values of this field are:

DFSC	Meaning
0b000100	Translation fault.
0b001100	Permission fault.
0b010000	Synchronous External abort, other than synchronous parity or ECC error.
0b010001	Asynchronous SError exception.
0b011000	Synchronous parity or ECC error on memory access.
0b011001	Asynchronous SError exception, from a parity or ECC error on memory access.
0b100001	Alignment fault.
0b100010	Debug exception, only when the EC value is 0b100100.
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive access).

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for Exception from a Data Abort

The following describe cases where Data Abort exceptions can be routed to Hyp mode, generating exceptions that are reported in the HSR with EC value 0b100100:

- ‘Abort exceptions, when the value of HCR.TGE is 1’.
- ‘Routing debug exceptions to EL2 using AArch32’.

The following describe cases that can cause a Data Abort exception that is taken to Hyp mode, and reported in the HSR with EC value of 0b100000 or 0b100100:

- ‘Hyp mode control of Non-secure access permissions’.
- ‘Memory fault reporting in Hyp mode’.

Accessing HSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HSR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0101	0b0010	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HSR() = R(t);
end;

```

D10.2.74 HSTR, Hyp System Trap Register

The HSTR characteristics are:

Purpose

Controls trapping to Hyp mode of Non-secure accesses, at EL1 or lower, to System registers in the coproc == 0b1111 encoding space:

- By the CRn value used to access the register using MCR or MRC instruction.
- By the CRm value used to access the register using MCRR or MRRC instruction.

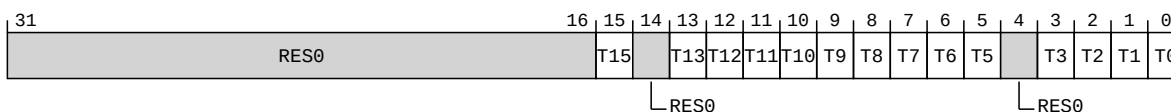
Configuration

There are no configuration notes.

Attributes

HSTR is a 32-bit register.

Field descriptions



Bits [31:16, 14, 4]

Reserved, RES0.

T<n> , bits [n], for n = 15 to 15, 13 to 5, 3 to 0

The remaining fields control whether Non-secure EL0 and EL1 accesses, using MCR or MRC instructions, reported using EC syndrome value 0x03, and MCRR or MRRC instructions, reported using EC syndrome value 0x04, to the System registers in the coproc == 0b1111 encoding space are trapped to Hyp mode:

T<n>	Meaning
0b0	This control has no effect on Non-secure EL0 or EL1 accesses to System registers.
0b1	Any Non-secure EL1 MCR or MRC access with coproc == 0b1111 and CRn == <n> is trapped to Hyp mode. A Non-secure EL0 MCR or MRC access with these values is trapped to Hyp mode only if the access is not UNDEFINED when the value of this field is 0. Any Non-secure EL1 MCRR or MRRC access with coproc == 0b1111 and CRm == <n> is trapped to Hyp mode. A Non-secure EL0 MCRR or MRRC access with these values is trapped to Hyp mode only if the access is not UNDEFINED when the value of this field is 0.

For example, when HSTR.T7 is 1, for instructions executed at Non-secure EL1:

- An MCR or MRC instruction with coproc set to 0b1111 and <CRn> set to c7 is trapped to Hyp mode.
- An MCRR or MRRC instruction with coproc set to 0b1111 and <CRm> set to c7 is trapped to Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to the expression 0x0000.

Accessing HSTR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b011

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HSTR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b011

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HSTR() = R(t);
end;

```

D10.2.75 HTPIDR, Hyp Software Thread ID Register

The HTPIDR characteristics are:

Purpose

Provides a location where software running in Hyp mode can store thread identifying information that is not visible to Non-secure software executing at EL0 or EL1, for hypervisor management purposes.

The PE makes no use of this register.

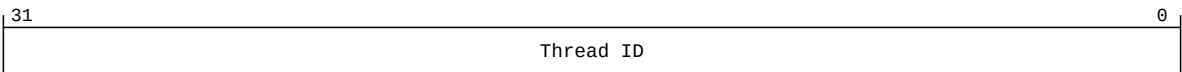
Configuration

Note
The PE never updates this register.

Attributes

HTPIDR is a 32-bit register.

Field descriptions



TID, bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HTPIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HTPIDR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HTPIDR() = R(t);
end;
```

D10.2.76 HVBAR, Hyp Vector Base Address Register

The HVBAR characteristics are:

Purpose

Holds the vector base address for any exception that is taken to Hyp mode.

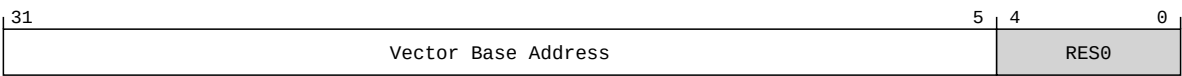
Configuration

There are no configuration notes.

Attributes

HVBAR is a 32-bit register.

Field descriptions



VBA, bits [31:5]

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:0]

Reserved, RES0.

Accessing HVBAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T12 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HVBAR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T12 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    HVBAR() = R(t);
end;
```

D10.2.77 ICIALLU, Instruction Cache Invalidate All to PoU

The ICIALLU characteristics are:

Purpose

Invalidate all instruction caches of the PE executing the instruction to the Point of Unification. If branch predictors are architecturally visible, also flush branch predictors.

Configuration

There are no configuration notes.

Attributes

ICIALLU is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing ICIALLU

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction.

When [HCR.FB](#) is 1, at Non-secure EL1 this instruction executes as a [ICIALLUIS](#).

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elseif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TPU == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(CacheOpScope_ALLU);
        end;
    end;
elseif PSTATE.EL == EL2 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
```



```
        else
            AArch32_IC(CacheOpScope_ALLU);
        end;
    end;
end;
```

D10.2.78 ICIALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The ICIALLUIS characteristics are:

Purpose

Invalidate all instruction caches in the Inner Shareable domain of the PE executing the instruction to the Point of Unification. If branch predictors are architecturally visible, also flush branch predictors.

Configuration

There are no configuration notes.

Attributes

ICIALLUIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing ICIALLUIS

The PE ignores the value of <Rt>. Software does not have to write a value to this register before issuing this instruction. Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TPU == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(CacheOpScope_ALLUIS);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(CacheOpScope_ALLUIS);
        end;
    end;
end;
```

D10.2.79 ICIMVAU, Instruction Cache line Invalidate by VA to PoU

The ICIMVAU characteristics are:

Purpose

Invalidate instruction cache line by virtual address to PoU.

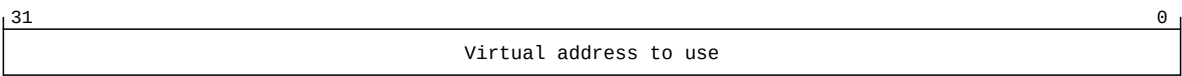
Configuration

There are no configuration notes.

Attributes

ICIMVAU is a 32-bit System instruction.

Field descriptions



VA, bits [31:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing ICIMVAU

Execution of this instruction might require an address translation from VA to PA, and that translation might fault.

For more information about faults, see [‘Permission fault’](#).

For more information about data cache maintenance instructions, see [‘AArch32 instruction cache maintenance instructions \(IC*\)’](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0101	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TPU == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch32_IC(R(t), CacheOpScope_PoU);
        end;
    end;
```

```
        end;  
    elsif PSTATE.EL == EL2 then  
        if !AArch32_CanTrapIC(CacheOp_Invalidate, CacheOpScope_PoU) then  
            ExecuteAsNOP();  
        else  
            if AArch32_TreatICAsNOP(CacheOp_Invalidate, CacheOpScope_PoU) then  
                ExecuteAsNOP();  
            else  
                AArch32_IC(R(t), CacheOpScope_PoU);  
            end;  
        end;  
    end;  
end;
```

D10.2.80 ID_AFR0, Auxiliary Feature Register 0

The ID_AFR0 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32 state.

Must be interpreted with the Main ID Register, [MIDR](#).

For general information about the interpretation of the ID registers see ‘[Principles of the ID scheme for fields in ID registers](#)’.

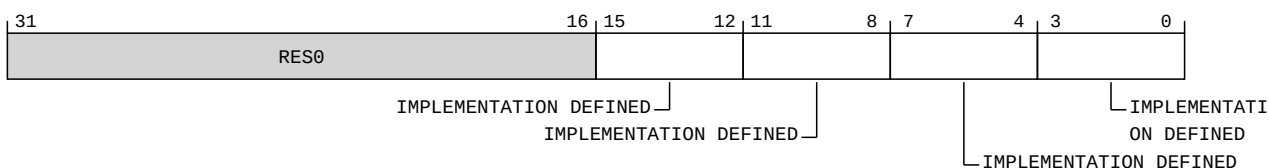
Configuration

There are no configuration notes.

Attributes

ID_AFR0 is a 32-bit register.

Field descriptions



Bits [31:16]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [3:0]

IMPLEMENTATION DEFINED.

Accessing ID_AFR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_AFR0();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = ID_AFR0();
end;
```

D10.2.81 ID_DFR0, Debug Feature Register 0

The ID_DFR0 characteristics are:

Purpose

Provides top-level information about the debug system in AArch32 state.

Must be interpreted with the Main ID Register, [MIDR](#).

For general information about the interpretation of the ID registers see ‘[Principles of the ID scheme for fields in ID registers](#)’.

Configuration

There are no configuration notes.

Attributes

ID_DFR0 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
RES0		PerfMon		MProfDbg		MMapTrc		CopTrc		MMapDbg		CopSDBG		CopDbg	

Bits [31:28]

Reserved, RES0.

PerfMon, bits [27:24]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in ‘[Alternative ID scheme used for the Performance Monitors Extension version](#)’.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

[FEAT_PMUv3](#) implements the functionality identified by the value 0b0011.

In any Armv8 implementation, the values 0b0001 and 0b0010 are not permitted.

Note

In Armv7, the value 0b0000 can mean that PMUv1 is implemented. PMUv1 and PMUv2 are not permitted in an Armv8 implementation.

Access to this field is RO.

MProfDbg, bits [23:20]

M-profile Debug. Support for memory-mapped debug model for M-profile processors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MProfDbg	Meaning
0b0000	Not supported.
0b0001	Support for M-profile Debug architecture, with memory-mapped access.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

MMapTrc, bits [19:16]

Memory-mapped Trace. Support for memory-mapped trace model.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MMapTrc	Meaning
0b0000	Not supported.
0b0001	Support for Arm trace architecture, with memory-mapped access.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

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

Access to this field is RO.

CopTrc, bits [15:12]

Support for System registers-based trace model, using registers in the coproc == 0b1110 encoding space.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CopTrc	Meaning
0b0000	Not supported.
0b0001	Support for Arm trace architecture, with System registers access.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

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

Access to this field is RO.

MMapDbg, bits [11:8]

Memory-mapped Debug. Support for Armv7 memory-mapped debug model for A and R-profile processors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MMapDbg	Meaning
0b0000	Not supported.
0b0100	Support for Armv7, v7 Debug architecture, with memory-mapped access.
0b0101	Support for Armv7, v7.1 Debug architecture, with memory-mapped access.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

The optional memory map defined by Armv8 is not compatible with Armv7.

Access to this field is RO.

CopSDBG, bits [7:4]

Support for a System registers-based Secure debug model, using registers in the coproc = 0b1110 encoding space, for an A-profile processor that includes EL3.

If EL3 is not implemented and the implemented Security state is Non-secure state, this field is RES0. Otherwise, this field reads the same as bits [3:0].

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CopDBG, bits [3:0]

Debug architecture version. Indicates presence of Armv8 debug architecture.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

The values 0b0000, 0b0010, 0b0011, 0b0100, and 0b0101 are not permitted in Armv8.

Access to this field is RO.

Accessing ID_DFR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b010

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_DFR0();
    end;
elsif PSTATE.EL == EL2 then

```

```
R(t) = ID_DFR0();  
end;
```

D10.2.82 ID_ISAR0, Instruction Set Attribute Register 0

The ID_ISAR0 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR1, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

Configuration

There are no configuration notes.

Attributes

ID_ISAR0 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
RES0		Divide		Debug		Coprocc		CmpBranch		BitField		BitCount		Swap	

Bits [31:28]

Reserved, RES0.

Divide, bits [27:24]

Indicates the implemented Divide instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Divide	Meaning
0b0000	None implemented.
0b0001	Adds SDIV and UDIV in the T32 instruction set.
0b0010	As for 0b0001, and adds SDIV and UDIV in the A32 instruction set.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

Debug, bits [23:20]

Indicates the implemented Debug instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Debug	Meaning
0b0000	None implemented.
0b0001	Adds BKPT.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Coproc, bits [19:16]

Indicates the implemented System register access instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Coproc	Meaning
0b0000	None implemented, except for instructions separately attributed by the architecture to provide access to AArch32 System registers and System instructions.
0b0001	Adds generic CDP, LDC, MCR, MRC, and STC.
0b0010	As for 0b0001, and adds generic CDP2, LDC2, MCR2, MRC2, and STC2.
0b0011	As for 0b0010, and adds generic MCRR and MRRC.
0b0100	As for 0b0011, and adds generic MCRR2 and MRRC2.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

CmpBranch, bits [15:12]

Indicates the implemented combined Compare and Branch instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CmpBranch	Meaning
0b0000	None implemented.
0b0001	Adds CBNZ and CBZ.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

BitField, bits [11:8]

Indicates support for BitField instructions BFC, BFI, SBFX, and UBFX.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BitField	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

BitCount, bits [7:4]

Indicates the implemented Bit Counting instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BitCount	Meaning
0b0000	None implemented.
0b0001	Adds CLZ.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Swap, bits [3:0]

Indicates the implemented Swap instructions in the A32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Swap	Meaning
0b0000	None implemented.
0b0001	Adds SWP and SWPB.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Accessing ID_ISAR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR0();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = ID_ISAR0();
end;
```

D10.2.83 ID_ISAR1, Instruction Set Attribute Register 1

The ID_ISAR1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR2, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

Configuration

There are no configuration notes.

Attributes

ID_ISAR1 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
Jazelle				Interwork				Immediate				IfThen			
								Extend				Except_AR			
												Except			
												Endian			

Jazelle, bits [31:28]

Indicates the implemented Jazelle extension instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Jazelle	Meaning
0b0000	No support for Jazelle.
0b0001	Adds the <code>BXJ</code> instruction, and the J bit in the PSR. This setting might indicate a trivial implementation of the Jazelle extension.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Interwork, bits [27:24]

Indicates the implemented Interworking instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Interwork	Meaning
0b0000	None implemented.

Interwork	Meaning
0b0001	Adds the BX instruction, and the T bit in the PSR.
0b0010	As for 0b0001, and adds the BLX instruction. PC loads have BX-like behavior.
0b0011	As for 0b0010, and guarantees that data-processing instructions in the A32 instruction set with the PC as the destination and the S bit clear have BX-like behavior.

All other values are reserved.

In Armv8, the only permitted value is 0b0011.

Access to this field is RO.

Immediate, bits [23:20]

Indicates support for data-processing instructions with long immediates:

- The `MOVT` instruction
- The `MOV` instruction encodings with zero-extended 16-bit immediates.
- The T32 `ADD` and `SUB` instruction encodings with zero-extended 12-bit immediates, and the other `ADD`, `ADR`, and `SUB` encodings cross-referenced by the pseudocode for those encodings.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Immediate	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

IfThen, bits [19:16]

Indicates the implemented If-Then instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IfThen	Meaning
0b0000	None implemented.
0b0001	Adds the IT instructions, and the IT bits in the PSRs.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Extend, bits [15:12]

Indicates the implemented Extend instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Extend	Meaning
0b0000	No scalar sign-extend or zero-extend instructions are implemented, where scalar instructions means non-Advanced SIMD instructions.
0b0001	Adds the SXTB, SXTB, UXTB, and UXTH instructions.
0b0010	As for 0b0001, and adds the SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

Except_AR, bits [11:8]

Indicates the implemented A and R-profile exception-handling instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Except_AR	Meaning
0b0000	None implemented.
0b0001	Adds the SRS and RFE instructions, and the A and R-profile forms of the CPS instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Except, bits [7:4]

Indicates the implemented exception-handling instructions in the A32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Except	Meaning
0b0000	Not implemented. This indicates that the User bank and Exception return forms of the LDM and STM instructions are not implemented.
0b0001	Adds the LDM (exception return), LDM (user registers), and STM (user registers) instruction versions.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Endian, bits [3:0]

Indicates the implemented Endian instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Endian	Meaning
0b0000	None implemented.
0b0001	Adds the SETEND instruction, and the E bit in the PSRs.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Accessing ID_ISAR1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR1();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR1();
end;

```

D10.2.84 ID_ISAR2, Instruction Set Attribute Register 2

The ID_ISAR2 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR3, ID_ISAR4, and ID_ISAR5.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

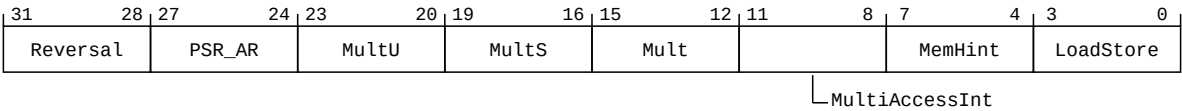
Configuration

There are no configuration notes.

Attributes

ID_ISAR2 is a 32-bit register.

Field descriptions



Reversal, bits [31:28]

Indicates the implemented Reversal instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Reversal	Meaning
0b0000	None implemented.
0b0001	Adds the REV, REV16, and REVSH instructions.
0b0010	As for 0b0001, and adds the RBIT instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

PSR_AR, bits [27:24]

Indicates the implemented A and R-profile instructions to manipulate the PSR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PSR_AR	Meaning
0b0000	None implemented.
0b0001	Adds the MRS and MSR instructions, and the exception return forms of data-processing instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

The exception return forms of the data-processing instructions are:

- In the A32 instruction set, data-processing instructions with the PC as the destination and the S bit set. These instructions might be affected by the WithShifts attribute.
- In the T32 instruction set, the SUBS PC, LR, #N instruction.

Access to this field is RO.

MultU, bits [23:20]

Indicates the implemented advanced unsigned Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultU	Meaning
0b0000	None implemented.
0b0001	Adds the UMULL and UMLAL instructions.
0b0010	As for 0b0001, and adds the UMAAL instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

MultS, bits [19:16]

Indicates the implemented advanced signed Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultS	Meaning
0b0000	None implemented.
0b0001	Adds the SMULL and SMLAL instructions.
0b0010	As for 0b0001, and adds the SMLABB, SMLABT, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, and SMULWT instructions. Also adds the Q bit in the PSRs.
0b0011	As for 0b0010, and adds the SMLAD, SMLADX, SMLALD, SMLALDX, SMLSD, SMLSDX, SMLSLD, SMLSLDX, SMMLA, SMMLAR, SMMLS, SMMLSR, SMMUL, SMMULR, SMUAD, SMUADX, SMUSD, and SMUSDX instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0011.

Access to this field is RO.

Mult, bits [15:12]

Indicates the implemented additional Multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Mult	Meaning
0b0000	No additional instructions implemented. This means only MUL is implemented.
0b0001	Adds the MLA instruction.
0b0010	As for 0b0001, and adds the MLS instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

MultiAccessInt, bits [11:8]

Indicates the support for interruptible multi-access instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MultiAccessInt	Meaning
0b0000	No support. This means the LDM and STM instructions are not interruptible.
0b0001	LDM and STM instructions are restartable.
0b0010	LDM and STM instructions are continuable.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

MemHint, bits [7:4]

Indicates the implemented Memory Hint instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MemHint	Meaning
0b0000	None implemented.
0b0001	Adds the PLD instruction.
0b0010	Adds the PLD instruction. (0b0001 and 0b0010 have identical effects.)

MemHint	Meaning
0b0011	As for 0b0001 (or 0b0010), and adds the <code>PLI</code> instruction.
0b0100	As for 0b0011, and adds the <code>PLDW</code> instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0100.

Access to this field is RO.

LoadStore, bits [3:0]

Indicates the implemented additional load/store instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LoadStore	Meaning
0b0000	No additional load/store instructions implemented.
0b0001	Adds the <code>LDRD</code> and <code>STRD</code> instructions.
0b0010	As for 0b0001, and adds the Load Acquire (<code>LDAB</code> , <code>LDAH</code> , <code>LDA</code> , <code>LDAEXB</code> , <code>LDAEXH</code> , <code>LDAEX</code> , <code>LDAEXD</code>) and Store Release (<code>STLB</code> , <code>STLH</code> , <code>STL</code> , <code>STLEXB</code> , <code>STLEXH</code> , <code>STLEX</code> , <code>STLEXD</code>) instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

Accessing ID_ISAR2

Accesses to this register use the following encodings in the System register encoding space:

`MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}`

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b010

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR2();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR2();
end;

```

D10.2.85 ID_ISAR3, Instruction Set Attribute Register 3

The ID_ISAR3 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with [ID_ISAR0](#), [ID_ISAR1](#), [ID_ISAR2](#), [ID_ISAR4](#), and [ID_ISAR5](#).

For general information about the interpretation of the ID registers see ‘[Principles of the ID scheme for fields in ID registers](#)’.

Configuration

There are no configuration notes.

Attributes

ID_ISAR3 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
T32EE		TrueNOP		T32Copy		TabBranch		SynchPrim		SVC		SIMD		Saturate	

T32EE, bits [31:28]

Indicates the implemented T32EE instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

T32EE	Meaning
0b0000	None implemented.
0b0001	Adds the <code>ENTERX</code> and <code>LEAVEX</code> instructions, and modifies the load behavior to include null checking.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

TrueNOP, bits [27:24]

Indicates the implemented true `NOP` instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TrueNOP	Meaning
0b0000	None implemented. This means there are no NOP instructions that do not have any register dependencies.
0b0001	Adds true NOP instructions in both the T32 and A32 instruction sets. This also permits additional NOP-compatible hints.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

T32Copy, bits [23:20]

Indicates the support for T32 non flag-setting MOV instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

T32Copy	Meaning
0b0000	Not supported. This means that in the T32 instruction set, encoding T1 of the MOV (register) instruction does not support a copy from a low register to a low register.
0b0001	Adds support for T32 instruction set encoding T1 of the MOV (register) instruction, copying from a low register to a low register.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

TabBranch, bits [19:16]

Indicates the implemented Table Branch instructions in the T32 instruction set.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TabBranch	Meaning
0b0000	None implemented.
0b0001	Adds the TBB and TBH instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

SynchPrim, bits [15:12]

Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented Synchronization Primitive instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SynchPrim	Meaning
0b0000	If SynchPrim_frac == 0b000, no Synchronization Primitives implemented.
0b0001	If SynchPrim_frac == 0b000, adds the LDREX and STREX instructions. If SynchPrim_frac == 0b011, also adds the CLREX, LDREXB, STREXB, and STREXH instructions.
0b0010	If SynchPrim_frac == 0b000, as for [0b001, 0b011] and also adds the LDREXD and STREXD instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

SVC, bits [11:8]

Indicates the implemented SVC instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SVC	Meaning
0b0000	Not implemented.
0b0001	Adds the SVC instruction.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

SIMD, bits [7:4]

Indicates the implemented SIMD instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMD	Meaning
0b0000	None implemented.
0b0001	Adds the SSAT and USAT instructions, and the Q bit in the PSRs.
0b0011	As for 0b0001, and adds the PKHBT, PKHTB, QADD16, QADD8, QASX, QSUB16, QSUB8, QSAX, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHSAX, SHSUB16, SHSUB8, SHSAX, SSAT16, SSUB16, SSUB8, SSAX, SXTAB16, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSUB16, UHSUB8, UHSAX, UQADD16, UQADD8, UQASX, UQSUB16, UQSUB8, UQSAX, USAD8, USADA8, USAT16, USUB16, USUB8, USAX, UXTAB16, and UXTB16 instructions. Also adds support for the GE[3:0] bits in the PSRs.

All other values are reserved.

In Armv8, the only permitted value is 0b0011.

The SIMD field relates only to implemented instructions that perform SIMD operations on the general-purpose registers. In an implementation that supports Advanced SIMD and floating-point instructions, [MVFR0](#), [MVFR1](#), and [MVFR2](#) give information about the implemented Advanced SIMD instructions.

Access to this field is RO.

Saturate, bits [3:0]

Indicates support for Saturate instructions `QADD`, `QDADD`, `QDSUB`, and `QSUB` instructions, and the Q bit in the PSRs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Saturate	Meaning
0b0000	None implemented. This means no non-Advanced SIMD saturate instructions are implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Accessing ID_ISAR3

Accesses to this register use the following encodings in the System register encoding space:

`MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}`

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b011

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR3();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR3();
end;

```

D10.2.86 ID_ISAR4, Instruction Set Attribute Register 4

The ID_ISAR4 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR5.

For general information about the interpretation of the ID registers, see ‘Principles of the ID scheme for fields in ID registers’.

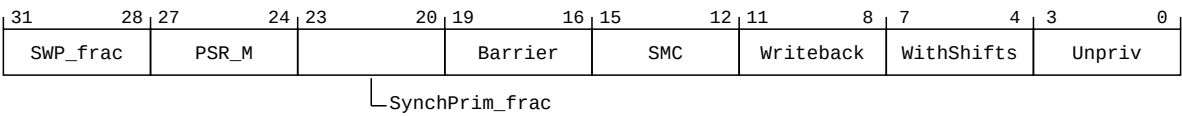
Configuration

There are no configuration notes.

Attributes

ID_ISAR4 is a 32-bit register.

Field descriptions



SWP_frac, bits [31:28]

Indicates support for the memory system locking the bus for SWP or SWPB instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SWP_frac	Meaning
0b0000	SWP or SWPB instructions not implemented.
0b0001	SWP or SWPB implemented but only in a uniprocessor context. SWP and SWPB do not guarantee whether memory accesses from other Requesters can come between the load memory access and the store memory access of the SWP or SWPB.

All other values are reserved. This field is valid only if ID_ISAR0.Swap is 0b0000.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

PSR_M, bits [27:24]

Indicates the implemented M-profile instructions to modify the PSRs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PSR_M	Meaning
0b0000	None implemented.
0b0001	Adds the M-profile forms of the CPS, MRS, and MSR instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

SynchPrim_frac, bits [23:20]

Used in conjunction with [ID_ISAR3.SynchPrim](#) to indicate the implemented Synchronization Primitive instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SynchPrim_frac	Meaning
0b0000	If SynchPrim == 0b0000, no Synchronization Primitives implemented. If SynchPrim == 0b0001, adds the LDREX and STREX instructions. If SynchPrim == 0b0010, also adds the CLREX, LDREXB, LDREXH, STREXB, STREXH, LDREXD, and STREXD instructions.
0b0011	If SynchPrim == 0b0001, adds the LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH instructions.

All other combinations of SynchPrim and SynchPrim_frac are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Barrier, bits [19:16]

Indicates support for Barrier instructions DMB, DSB, and ISB in the T32 and A32 instruction sets.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Barrier	Meaning
0b0000	None implemented. Barrier operations are provided only as System instructions in the (coproc==0b1111) encoding space.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

SMC, bits [15:12]

Indicates the implemented SMC instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SMC	Meaning
0b0000	None implemented.
0b0001	Adds the SMC instruction.

All other values are reserved.

In Armv8-R AArch32, the only permitted value is 0b0001.

Access to this field is RO.

Writeback, bits [11:8]

Indicates the support for Writeback addressing modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Writeback	Meaning
0b0000	Basic support. Only the LDM, STM, PUSH, POP, SRS, and RFE instructions support writeback addressing modes. These instructions support all of their writeback addressing modes.
0b0001	Adds support for all of the writeback addressing modes.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

WithShifts, bits [7:4]

Indicates the support for instructions with shifts.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WithShifts	Meaning
0b0000	Nonzero shifts supported only in MOV and shift instructions.
0b0001	Adds support for shifts of loads and stores over the range LSL 0-3.
0b0011	As for 0b0001, and adds support for other constant shift options, both on load/store and other instructions.
0b0100	As for 0b0011, and adds support for register-controlled shift options.

All other values are reserved.

In Armv8, the only permitted value is 0b0100.

Access to this field is RO.

Unpriv, bits [3:0]

Indicates the implemented unprivileged instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Unpriv	Meaning
0b0000	None implemented. No T variant instructions are implemented.
0b0001	Adds the LDRBT, LDRT, STRBT, and STRT instructions.
0b0010	As for 0b0001, and adds the LDRHT, LDRSBT, LDRSHT, and STRHT instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

Accessing ID_ISAR4

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR4();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR4();
end;
```

D10.2.87 ID_ISAR5, Instruction Set Attribute Register 5

The ID_ISAR5 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0, ID_ISAR1, ID_ISAR2, ID_ISAR3, and ID_ISAR4.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

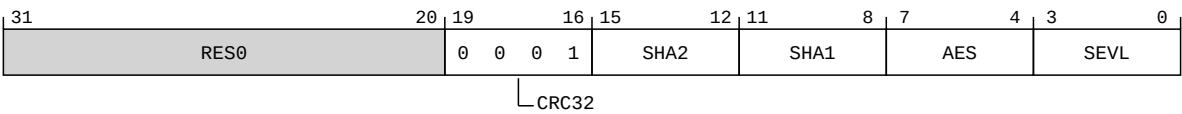
Configuration

There are no configuration notes.

Attributes

ID_ISAR5 is a 32-bit register.

Field descriptions



Bits [31:20]

Reserved, RES0.

CRC32, bits [19:16]

Indicates support for the CRC32 instructions CRC32B, CRC32H, CRC32W, CRC32CB, CRC32CH, and CRC32CW in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CRC32	Meaning
0b0000	CRC32 instructions are not implemented.
0b0001	The specified instructions are implemented.

The table above shows all possible values of this field. 0b0001 is the value of this field in the current implementation.

All other values are reserved.

FEAT_CRC32 implements the functionality identified by the value 0b0001.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

SHA2, bits [15:12]

Indicates support for the SHA2 instructions SHA256H, SHA256H2, SHA256SU0, and SHA256SU1 in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA2	Meaning
0b0000	No SHA2 instructions implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SHA1, bits [11:8]

Indicates support for the SHA1 instructions SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA1	Meaning
0b0000	No SHA1 instructions implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

AES, bits [7:4]

Indicates support for the AES instructions in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AES	Meaning
0b0000	No AES instructions implemented.
0b0001	AESE, AESD, AESMC, and AESIMC implemented.
0b0010	As for 0b0001, plus VMULL (polynomial) instructions operating on 64-bit data quantities.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0010.

Access to this field is RO.

SEVL, bits [3:0]

Indicates support for the SEVL instruction in AArch32 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SEVL	Meaning
0b0000	SEVL is implemented as a NOP.
0b0001	SEVL is implemented as Send Event Local.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Accessing ID_ISAR5

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_ISAR5();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_ISAR5();
end;
```

D10.2.88 ID_MMFR0, Memory Model Feature Register 0

The ID_MMFR0 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

Configuration

There are no configuration notes.

Attributes

ID_MMFR0 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
InnerShr		FCSE		AuxReg		TCM		ShareLvl		OuterShr		PMSA		VMSA	

InnerShr, bits [31:28]

Innermost Shareability. Indicates the innermost shareability domain implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

InnerShr	Meaning
0b0000	Implemented as Non-cacheable.
0b0001	Implemented with hardware coherency support.
0b1111	Shareability ignored.

All other values are reserved.

In Armv8, the permitted values are 0b0000, 0b0001, and 0b1111.

This field is valid only if the implementation supports two levels of shareability, as indicated by ID_MMFR0.ShareLvl having the value 0b0001.

When ID_MMFR0.ShareLvl is zero, this field is UNKNOWN.

Access to this field is RO.

FCSE, bits [27:24]

Indicates whether the implementation includes the FCSE.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FCSE	Meaning
0b0000	Not supported.
0b0001	Support for FCSE.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

AuxReg, bits [23:20]

Auxiliary Registers. Indicates support for Auxiliary registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AuxReg	Meaning
0b0000	None supported.
0b0001	Support for Auxiliary Control Register only.
0b0010	Support for Auxiliary Fault Status Registers (AIFSR and ADFSR) and Auxiliary Control Register.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Note

Accesses to unimplemented Auxiliary registers are UNDEFINED.

Access to this field is RO.

TCM, bits [19:16]

Indicates support for TCMs and associated DMAs.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TCM	Meaning
0b0000	Not supported.
0b0001	Support is IMPLEMENTATION DEFINED.
0b0010	Support for TCM only, Armv6 implementation.
0b0011	Support for TCM and DMA, Armv6 implementation.

All other values are reserved.

In Armv8-R AArch32, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

ShareLvl, bits [15:12]

Shareability Levels. Indicates the number of shareability levels implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ShareLvl	Meaning
0b0000	One level of shareability implemented.
0b0001	Two levels of shareability implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

OuterShr, bits [11:8]

Outermost Shareability. Indicates the outermost shareability domain implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OuterShr	Meaning
0b0000	Implemented as Non-cacheable.
0b0001	Implemented with hardware coherency support.
0b1111	Shareability ignored.

All other values are reserved.

In Armv8, the permitted values are 0b0000, 0b0001, and 0b1111.

Access to this field is RO.

PMSA, bits [7:4]

Indicates support for a PMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMSA	Meaning
0b0000	Not supported.
0b0001	Support for IMPLEMENTATION DEFINED PMSA.
0b0010	Support for PMSAv6, with a Cache Type Register implemented.
0b0011	Support for PMSAv7, with support for memory subsections. Armv7-R profile.
0b0100	Support for PMSAv8-32, with base and limit. Armv8-R AArch32 profile.

All other values are reserved.

In Armv8-R AArch32, the only permitted value is 0b0100.

Access to this field is RO.

VMSA, bits [3:0]

Indicates support for a VMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMSA	Meaning
0b0000	Not supported.
0b0001	Support for IMPLEMENTATION DEFINED VMSA.
0b0010	Support for VMSAv6, with Cache and TLB Type Registers implemented.
0b0011	Support for VMSAv7, with support for remapping and the Access flag. ARMv7-A profile.
0b0100	As for 0b0011, and adds support for the PXN bit in the Short-descriptor translation table format descriptors.
0b0101	As for 0b0100, and adds support for the Long-descriptor translation table format.

All other values are reserved.

In Armv8-R AArch32, the only permitted value is 0b0000.

Access to this field is RO.

Accessing ID_MMFR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR0();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_MMFR0();
end;

```

D10.2.89 ID_MMFR1, Memory Model Feature Register 1

The ID_MMFR1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

Configuration

There are no configuration notes.

Attributes

ID_MMFR1 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
BPred				L1TstCln				L1Uni				L1Hvd			
								L1UniSW				L1HvdSW			
												L1UniVA			
												L1HvdVA			

BPred, bits [31:28]

Branch Predictor. Indicates branch predictor management requirements.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BPred	Meaning
0b0000	No branch predictor, or no MMU present. Implies a fixed MPU configuration.
0b0001	Branch predictor requires flushing on: - Enabling or disabling a stage of address translation. - Writing new data to instruction locations. - Writing new mappings to the translation tables. - Changes to the TTBR0, TTBR1, or TTBCR registers. - Changes to the ContextID or ASID, or to the FCSE ProcessID if this is supported.
0b0010	Branch predictor requires flushing on: - Enabling or disabling a stage of address translation. - Writing new data to instruction locations. - Writing new mappings to the translation tables.
0b0011	Branch predictor requires flushing only on writing new data to instruction locations.
0b0100	For execution correctness, branch predictor requires no flushing at any time.

All other values are reserved.

In Armv8, the permitted values are 0b0010, 0b0011, or 0b0100. For values other than 0b0000 and 0b0100, the Arm Architecture Reference Manual, or the product documentation, might give more information about the required maintenance.

Access to this field is RO.

L1TstCln, bits [27:24]

Level 1 cache Test and Clean. Indicates the supported Level 1 data cache test and clean operations, for Harvard or unified cache implementations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1TstCln	Meaning
0b0000	None supported.
0b0001	Supported Level 1 data cache test and clean operations are:
	Test and clean data cache.
0b0010	As for 0b0001, and adds:
	Test, clean, and invalidate data cache.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1Uni, bits [23:20]

Level 1 Unified cache. Indicates the supported entire Level 1 cache maintenance operations for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Uni	Meaning
0b0000	None supported.
0b0001	Supported entire Level 1 cache operations are:
	- Invalidate cache, including branch predictor if appropriate. - Invalidate branch predictor, if appropriate.
0b0010	As for 0b0001, and adds:
	- Clean cache, using a recursive model that uses the cache dirty status bit. - Clean and invalidate cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1Hvd, bits [19:16]

Level 1 Harvard cache. Indicates the supported entire Level 1 cache maintenance operations for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Hvd	Meaning
0b0000	None supported.
0b0001	Supported entire Level 1 cache operations are: • Invalidate instruction cache, including branch predictor if appropriate. • Invalidate branch predictor, if appropriate.
0b0010	As for 0b0001, and adds: • Invalidate data cache. • Invalidate data cache and instruction cache, including branch predictor if appropriate.
0b0011	As for 0b0010, and adds: • Clean data cache, using a recursive model that uses the cache dirty status bit. • Clean and invalidate data cache, using a recursive model that uses the cache dirty status bit.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1UniSW, bits [15:12]

Level 1 Unified cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1UniSW	Meaning
0b0000	None supported.
0b0001	Supported Level 1 unified cache line maintenance operations by set/way are: • Clean cache line by set/way.
0b0010	As for 0b0001, and adds: • Clean and invalidate cache line by set/way.
0b0011	As for 0b0010, and adds: • Invalidate cache line by set/way.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1HvdSW, bits [11:8]

Level 1 Harvard cache by Set/Way. Indicates the supported Level 1 cache line maintenance operations by set/way, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdSW	Meaning
0b0000	None supported.
0b0001	Supported Level 1 Harvard cache line maintenance operations by set/way are: - Clean data cache line by set/way. - Clean and invalidate data cache line by set/way.
0b0010	As for 0b0001, and adds: Invalidate data cache line by set/way.
0b0011	As for 0b0010, and adds: Invalidate instruction cache line by set/way.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1UniVA, bits [7:4]

Level 1 Unified cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a unified cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1UniVA	Meaning
0b0000	None supported.
0b0001	Supported Level 1 unified cache line maintenance operations by VA are: - Clean cache line by VA. - Invalidate cache line by VA. - Clean and invalidate cache line by VA.
0b0010	As for 0b0001, and adds: Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1HvdVA, bits [3:0]

Level 1 Harvard cache by Virtual Address. Indicates the supported Level 1 cache line maintenance operations by VA, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdVA	Meaning
0b0000	None supported.
0b0001	Supported Level 1 Harvard cache line maintenance operations by VA are: - Clean data cache line by VA. - Invalidate data cache line by VA. - Clean and invalidate data cache line by VA. - Clean instruction cache line by VA.
0b0010	As for 0b0001, and adds: Invalidate branch predictor by VA, if branch predictor is implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Accessing ID_MMFR1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b101

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR1();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_MMFR1();
end;

```

D10.2.90 ID_MMFR2, Memory Model Feature Register 2

The ID_MMFR2 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see [‘Principles of the ID scheme for fields in ID registers’](#).

Configuration

There are no configuration notes.

Attributes

ID_MMFR2 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
HWAACFlg				WFIStall				MemBarr				UniTLB			

0b0001

Support for WFI stalling.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

MemBarr, bits [23:20]

Memory Barrier. Indicates the supported memory barrier System instructions in the (coproc == 1111) encoding space.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MemBarr	Meaning
0b0000	None supported.
0b0001	Supported memory barrier System instructions are: • Data Synchronization Barrier (DSB).
0b0010	As for 0b0001, and adds: • Instruction Synchronization Barrier (ISB). • Data Memory Barrier (DMB).

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Arm deprecates the use of these operations. [ID_ISAR4.Barrier_instrs](#) indicates the level of support for the preferred barrier instructions.

Access to this field is RO.

UniTLB, bits [19:16]

Unified TLB. Indicates the supported TLB maintenance operations, for a unified TLB implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

UniTLB	Meaning
0b0000	Not supported.
0b0001	Supported unified TLB maintenance operations are: • Invalidate all entries in the TLB. • Invalidate TLB entry by VA.
0b0010	As for 0b0001, and adds: • Invalidate TLB entries by ASID match.

UniTLB	Meaning
0b0011	As for 0b0010, and adds: • Invalidate instruction TLB and data TLB entries by VA All ASID. This is a shared unified TLB operation
0b0100	As for 0b0011, and adds: • Invalidate Hyp mode unified TLB entry by VA. • Invalidate entire Non-secure PL1&0 unified TLB. • Invalidate entire Hyp mode unified TLB.
0b0101	As for 0b0100, and adds the following operations: TLBIMVALIS , TLBIMVAALIS , TLBIMVALHIS , TLBIMVAL , TLBIMVAAL , TLBIMVALH .
0b0110	As for 0b0101, and adds the following operations: TLBIIPAS2IS , TLBIIPAS2LIS , TLBIIPAS2 , TLBIIPAS2L .

All other values are reserved.

In Armv8-R AArch32, the only permitted value is 0b0000.

Access to this field is RO.

HvdTLB, bits [15:12]

If the value of ID_MMFR2.UniTLB is not 0b0000, then the meaning of this field is IMPLEMENTATION DEFINED. Arm deprecates the use of this field by software.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

L1HvdRng, bits [11:8]

Level 1 Harvard cache Range. Indicates the supported Level 1 cache maintenance range operations, for a Harvard cache implementation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdRng	Meaning
0b0000	Not supported.
0b0001	Supported Level 1 Harvard cache maintenance range operations are: • Invalidate data cache range by VA. • Invalidate instruction cache range by VA. • Clean data cache range by VA. • Clean and invalidate data cache range by VA.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1HvdBG, bits [7:4]

Level 1 Harvard cache Background fetch. Indicates the supported Level 1 cache background fetch operations, for a Harvard cache implementation. When supported, background fetch operations are non-blocking operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdBG	Meaning
0b0000	Not supported.
0b0001	Supported Level 1 Harvard cache background fetch operations are: - Fetch instruction cache range by VA. - Fetch data cache range by VA.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

L1HvdFG, bits [3:0]

Level 1 Harvard cache Foreground fetch. Indicates the supported Level 1 cache foreground fetch operations, for a Harvard cache implementation. When supported, foreground fetch operations are blocking operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1HvdFG	Meaning
0b0000	Not supported.
0b0001	Supported Level 1 Harvard cache foreground fetch operations are: - Fetch instruction cache range by VA. - Fetch data cache range by VA.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

Accessing ID_MMFR2

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b110

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    
```

```
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR2();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_MMFR2();
end;
```

D10.2.91 ID_MMFR3, Memory Model Feature Register 3

The ID_MMFR3 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

Configuration

There are no configuration notes.

Attributes

ID_MMFR3 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
Supersec		CMemSz		CohWalk		RES0		MaintBcst		BPMaint		CMaintSW		CMaintVA	

Supersec, bits [31:28]

Supersections. On a VMSA implementation, indicates whether Supersections are supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Supersec	Meaning
0b0000	Supersections supported.
0b1111	Supersections not supported.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b1111.

Access to this field is RO.

CMemSz, bits [27:24]

Cached Memory Size. Indicates the physical memory size supported by the caches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CMemSz	Meaning
0b0000	4GB, corresponding to a 32-bit physical address range.
0b0001	64GB, corresponding to a 36-bit physical address range.

0b0010

1TB or more, corresponding to a 40-bit or larger physical address range.

All other values are reserved.

In Armv8, the permitted values are 0b0000, 0b0001, and 0b0010.

Access to this field is RO.

CohWalk, bits [23:20]

Coherent Walk. Indicates whether Translation table updates require a clean to the Point of Unification.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CohWalk	Meaning
0b0000	Updates to the translation tables require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.
0b0001	Updates to the translation tables do not require a clean to the Point of Unification to ensure visibility by subsequent translation table walks.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Bits [19:16]

Reserved, RES0.

MaintBcst, bits [15:12]

Maintenance Broadcast. Indicates whether Cache, TLB, and branch predictor operations are broadcast.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MaintBcst	Meaning
0b0000	Cache, TLB, and branch predictor operations only affect local structures.
0b0001	Cache and branch predictor operations affect structures according to shareability and defined behavior of instructions. TLB operations only affect local structures.
0b0010	Cache, TLB, and branch predictor operations affect structures according to shareability and defined behavior of instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

BPMaint, bits [11:8]

Branch Predictor Maintenance. Indicates the supported branch predictor maintenance operations in an implementation with hierarchical cache maintenance operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BPMaint	Meaning
0b0000	None supported.
0b0001	Supported branch predictor maintenance operations are: • Invalidate all branch predictors.
0b0010	As for 0b0001, and adds: • Invalidate branch predictors by VA.

All other values are reserved.

In Armv8, the only permitted value is 0b0010.

Access to this field is RO.

CMaintSW, bits [7:4]

Cache Maintenance by Set/Way. Indicates the supported cache maintenance operations by set/way, in an implementation with hierarchical caches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CMaintSW	Meaning
0b0000	None supported.
0b0001	Supported hierarchical cache maintenance instructions by set/way are: • Invalidate data cache by set/way. • Clean data cache by set/way. • Clean and invalidate data cache by set/way.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

In a unified cache implementation, the data cache maintenance operations apply to the unified caches.

Access to this field is RO.

CMaintVA, bits [3:0]

Cache Maintenance by Virtual Address. Indicates the supported cache maintenance operations by VA, in an implementation with hierarchical caches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CMaintVA	Meaning
0b0000	None supported.

CMaintVA	Meaning
0b0001	Supported hierarchical cache maintenance operations by VA are: • Invalidate data cache by VA. • Clean data cache by VA. • Clean and invalidate data cache by VA. • Invalidate instruction cache by VA. • Invalidate all instruction cache entries.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

In a unified cache implementation, data cache maintenance operations apply to the unified caches, and the instruction cache maintenance instructions are not implemented.

Access to this field is RO.

Accessing ID_MMFR3

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b111

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR3();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_MMFR3();
end;

```

D10.2.92 ID_MMFR4, Memory Model Feature Register 4

The ID_MMFR4 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

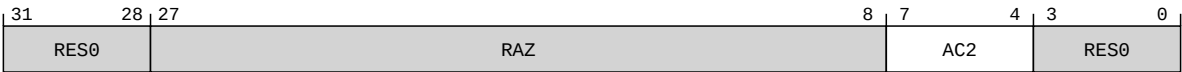
Configuration

There are no configuration notes.

Attributes

ID_MMFR4 is a 32-bit register.

Field descriptions



Bits [31:28]

Reserved, RES0.

Bits [27:8]

Reserved, RAZ.

AC2, bits [7:4]

Indicates the extension of the ACTLR and HACTLR registers using ACTLR2 and HACTLR2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AC2	Meaning
0b0000	ACTLR2 and HACTLR2 are not implemented.
0b0001	ACTLR2 and HACTLR2 are implemented.

All other values are reserved.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.2, the only permitted value is 0b0001.

Access to this field is RO.

Bits [3:0]

Reserved, RES0.

Accessing ID_MMFR4

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0010	0b110

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && (!IsZero(ID_MMFR4()) || ImpDefBool("ID_MMFR4 trapped by HCR.TID3")) &&
        ↪HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_MMFR4();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = ID_MMFR4();
end;
```

D10.2.93 ID_PFR0, Processor Feature Register 0

The ID_PFR0 characteristics are:

Purpose

Gives top-level information about the instruction sets and other features supported by the PE in AArch32 state.

Must be interpreted with ID_PFR1.

For general information about the interpretation of the ID registers, see ‘Principles of the ID scheme for fields in ID registers’.

Configuration

There are no configuration notes.

Attributes

ID_PFR0 is a 32-bit register.

Field descriptions

31	16	15	12	11	8	7	4	3	0	
RES0			State3		State2		State1		State0	

Bits [31:16]

Reserved, RES0.

State3, bits [15:12]

T32EE instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State3	Meaning
0b0000	Not implemented.
0b0001	T32EE instruction set implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

State2, bits [11:8]

Jazelle extension support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State2	Meaning
0b0000	Not implemented.
0b0001	Jazelle extension implemented, without clearing of JOSCR.CV on exception entry.
0b0010	Jazelle extension implemented, with clearing of JOSCR.CV on exception entry.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

State1, bits [7:4]

T32 instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State1	Meaning
0b0000	T32 instruction set not implemented.
0b0001	T32 encodings before the introduction of Thumb-2 technology implemented: • All instructions are 16-bit. • A BL or BLX is a pair of 16-bit instructions. • 32-bit instructions other than BL and BLX cannot be encoded.
0b0011	T32 encodings after the introduction of Thumb-2 technology implemented, for all 16-bit and 32-bit T32 basic instructions.

All other values are reserved.

In Armv8, the only permitted value is 0b0011.

Access to this field is RO.

State0, bits [3:0]

A32 instruction set support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

State0	Meaning
0b0000	A32 instruction set not implemented.
0b0001	A32 instruction set implemented.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Accessing ID_PFR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_PFR0();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_PFR0();
end;

```


D10.2.94 ID_PFR1, Processor Feature Register 1

The ID_PFR1 characteristics are:

Purpose

Gives information about the AArch32 programmers’ model.

Must be interpreted with ID_PFR0.

For general information about the interpretation of the ID registers see ‘Principles of the ID scheme for fields in ID registers’.

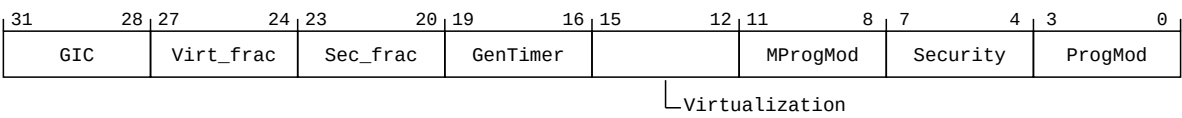
Configuration

There are no configuration notes.

Attributes

ID_PFR1 is a 32-bit register.

Field descriptions



GIC, bits [31:28]

System register GIC CPU interface.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GIC	Meaning
0b0000	GIC CPU interface System registers not implemented.
0b0001	System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
0b0011	System register interface to version 4.1 of the GIC CPU interface is supported.

The table above shows all possible values of this field. The values this field can take in the current implementation are:

- 0b0000
- 0b0001

All other values are reserved.

In Armv8-R AArch32, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Virt_frac, bits [27:24]

Virtualization fractional field. When the Virtualization field is 0b0000, determines the support for Virtualization Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Virt_frac	Meaning
0b0000	No Virtualization Extensions are implemented.
0b0001	The following Virtualization Extensions are implemented: • The MSR (banked register) and MRS (banked register) instructions. • The ERET instruction.

All other values are reserved.

In Armv8-R AArch32 the permitted value is 0b0000.

This field is valid only when the value of ID_PFR1.Virtualization is 0, otherwise it holds the value 0b0000.

Note

The ID_ISAR registers do not identify whether the instructions added by the Virtualization Extensions are implemented.

Access to this field is RO.

Sec_frac, bits [23:20]

Security fractional field. When the Security field is 0b0000, determines the support for Security Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Sec_frac	Meaning
0b0000	No Security Extensions are implemented.
0b0001	The following Security Extensions are implemented: • The VBAR register.
0b0010	As for 0b0001, and the ability to access Secure or Non-secure physical memory is supported.

All other values are reserved.

In Armv8-R AArch32 the permitted value is 0b0001.

This field is valid only when the value of ID_PFR1.Security is 0, otherwise it holds the value 0b0000.

Access to this field is RO.

GenTimer, bits [19:16]

Generic Timer support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GenTimer	Meaning
0b0000	Generic Timer is not implemented.
0b0001	Generic Timer is implemented.
0b0010	Generic Timer is implemented, and also includes support for CNTHCTL.EVTIS and CNTKCTL.EVTIS fields, and CNTPCTSS and CNTVCTSS counter views.

All other values are reserved.

FEAT_ECV implements the functionality identified by the value 0b0010.

In Armv8.0, the only permitted value is 0b0001.

From Armv8.6, the only permitted value is 0b0010.

Access to this field is RO.

Virtualization, bits [15:12]

Virtualization support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Virtualization	Meaning
0b0000	EL2, Hyp mode, and the HVC instruction not implemented.
0b0001	EL2, Hyp mode, and the HVC instruction.

All other values are reserved.

In Armv8-R AArch32 the permitted value is 0b0001.

Note

The ID_ISARs do not identify whether the HVC instruction is implemented.

Access to this field is RO.

MProMod, bits [11:8]

M-profile programmers' model support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MProMod	Meaning
0b0000	Not supported.
0b0010	Support for two-stack programmers' model.

All other values are reserved.

In Armv8-A and Armv8-R, the only permitted value is 0b0000.

Access to this field is RO.

Security, bits [7:4]

Security support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Security	Meaning
0b0000	EL3, Monitor mode, and the SMC instruction not implemented.
0b0001	EL3, Monitor mode, the SMC instruction, and all the features described by Sec_frac == 0b0001 implemented.
0b0010	As for 0b0001, and adds the ability to set the NSACR.RFR bit. Not permitted in Armv8 as the NSACR.RFR bit is RES0.

All other values are reserved.

In Armv8-R AArch32 the permitted value is 0b0000.

Access to this field is RO.

ProgMod, bits [3:0]

Support for the standard programmers' model for ARMv4 and later. Model must support User, FIQ, IRQ, Supervisor, Abort, Undefined, and System modes.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ProgMod	Meaning
0b0000	Not supported.
0b0001	Supported.

All other values are reserved.

In Armv8, the only permitted value is 0b0001.

Access to this field is RO.

Accessing ID_PFR1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0001	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);

```

```
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ID_PFR1();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ID_PFR1();
end;
```

D10.2.95 IFAR, Instruction Fault Address Register

The IFAR characteristics are:

Purpose

Holds the virtual address of the faulting address that caused a synchronous Prefetch Abort exception.

Configuration

There are no configuration notes.

Attributes

IFAR is a 32-bit register.

This register has the following instances:

- IFAR.

Field descriptions

31	0
VA of faulting address of synchronous Prefetch Abort exception	

VA, bits [31:0]

VA of faulting address of synchronous Prefetch Abort exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing IFAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = IFAR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = IFAR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        IFAR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    IFAR() = R(t);
end;
```

D10.2.96 IFSR, Instruction Fault Status Register

The IFSR characteristics are:

Purpose

Holds status information about the last instruction fault.

Configuration

The current translation table format determines which format of the register is used.

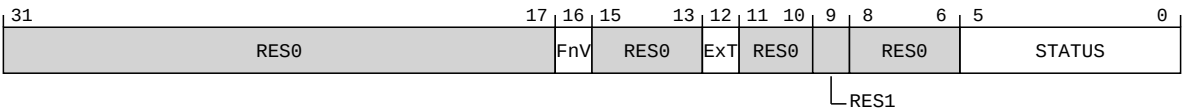
Attributes

IFSR is a 32-bit register.

This register has the following instances:

- IFSR.

Field descriptions



Bits [31:17]

Reserved, RES0.

FnV, bit [16]

FAR not Valid, for a synchronous External abort.

FnV	Meaning
0b0	IFAR is valid.
0b1	IFAR is not valid, and holds an UNKNOWN value.

This field is valid only for a synchronous External abort. It is RES0 for all other Prefetch Abort exceptions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:13]

Reserved, RES0.

ExT, bit [12]

External abort type. This bit can be used to provide an IMPLEMENTATION DEFINED classification of External aborts.

In an implementation that does not provide any classification of External aborts, this bit is RES0.

For aborts other than External aborts this bit always returns 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Bit [9]

Reserved, RES1.

Bits [8:6]

Reserved, RES0.

STATUS, bits [5:0]

Fault status bits. Possible values of this field are:

STATUS	Meaning
0b000100	Translation fault.
0b001100	Permission fault.
0b010000	Synchronous External abort, other than synchronous parity or ECC error.
0b011000	Synchronous parity or ECC error on memory access.
0b100001	PC alignment fault.
0b100010	Debug exception.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing IFSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = IFSR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = IFSR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T5 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        IFSR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    IFSR() = R(t);
end;
```

D10.2.97 **ISR, Interrupt Status Register**

The ISR characteristics are:

Purpose

Shows the pending status of the IRQ and FIQ interrupts and the SError exceptions.

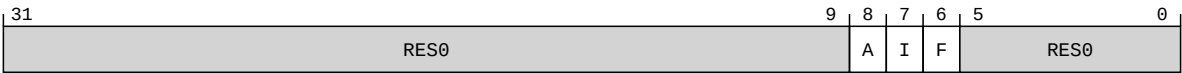
Configuration

There are no configuration notes.

Attributes

ISR is a 32-bit register.

Field descriptions



Bits [31:9]

Reserved, RES0.

A, bit [8]

SError exception pending bit:

A	Meaning
0b0	No pending SError exception.
0b1	An SError exception is pending.

If all of the following apply then this field shows the pending status of virtual SError exceptions:

- [HCR.AMO](#) is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical SError exceptions.

If the SError exception is edge-triggered, this field is cleared to zero when the physical SError exception is taken.

I, bit [7]

IRQ pending bit. Indicates whether an IRQ interrupt is pending:

I	Meaning
0b0	No pending IRQ.
0b1	An IRQ interrupt is pending.

If all of the following apply then this field shows the pending status of virtual IRQ interrupts:

- [HCR.IMO](#) is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical IRQ interrupts.

F, bit [6]

FIQ pending bit. Indicates whether an FIQ interrupt is pending.

F	Meaning
0b0	No pending FIQ.
0b1	An FIQ interrupt is pending.

If all of the following apply then this field shows the pending status of virtual FIQ interrupts:

- [HCR.FMO](#) is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical FIQ interrupts.

Bits [5:0]

Reserved, RES0.

Accessing ISR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0001	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T12 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = ISR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = ISR();
end;

```

D10.2.98 ITLBIALL, Instruction TLB Invalidate All

The ITLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
 - Would be required for the EL1&0 translation regime.
 - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the Non-secure PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This instruction is treated as a NOP.

Attributes

ITLBIALL is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing ITLBIALL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_ITLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
    end;
elseif PSTATE.EL == EL2 then
    AArch32_ITLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
end;
```

D10.2.99 ITLBIASID, Instruction TLB Invalidate by ASID match

The ITLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
 - Is from a level of lookup above the final level.
 - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

Configuration

This instruction is treated as a NOP.

Attributes

ITLBIASID is a 32-bit System instruction.

Field descriptions



Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing ITLBIASID

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_ITLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
            ↪R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_ITLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr, R(t));
end;
```

D10.2.100 ITLBIMVA, Instruction TLB Invalidate by VA

The ITLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from instruction TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
 - The entry is from a level of lookup above the final level and matches the specified ASID.
 - The entry is a global entry from the final level of lookup.
 - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Arm deprecates the use of this System instruction. It is only provided for backward compatibility with earlier versions of the Arm architecture.

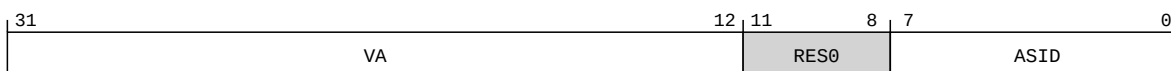
Configuration

This instruction is treated as a NOP.

Attributes

ITLBIMVA is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this operation, regardless of the value of the ASID field.

Executing ITLBIMVA

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0101	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_ITLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_ITLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```

D10.2.101 JIDR, Jazelle ID Register

The JIDR characteristics are:

Purpose

A Jazelle register, which identified the Jazelle architecture version.

Configuration

There are no configuration notes.

Attributes

JIDR is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RAZ.

Accessing JIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if ImpDefBool("JIDR UNDEFINED at EL0") then
        Undefined();
    elseif EL2Enabled() && HCR().TID0 == '1' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = JIDR();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR().TID0 == '1' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = JIDR();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = JIDR();
end;
```

D10.2.102 **JMCR, Jazelle Main Configuration Register**

The JMCR characteristics are:

Purpose

A Jazelle register, which provides control of the Jazelle extension.

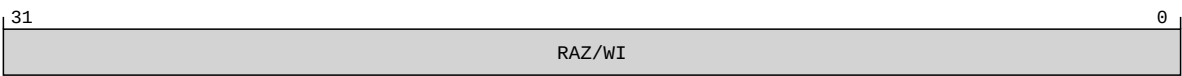
Configuration

There are no configuration notes.

Attributes

JMCR is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RAZ/WI.

Accessing JMCR

For accesses from EL0 it is IMPLEMENTATION DEFINED whether the register is RW or UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if ImpDefBool("JMCR UNDEFINED at EL0") then
        Undefined();
    else
        R(t) = JMCR();
    end;
elsif PSTATE.EL == EL1 then
    R(t) = JMCR();
elsif PSTATE.EL == EL2 then
    R(t) = JMCR();
end;
```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if ImpDefBool("JMCR UNDEFINED at EL0") then
        Undefined();
    else
        return;
    end;
elseif PSTATE.EL == EL1 then
    return;
elseif PSTATE.EL == EL2 then
    return;
end;
```

D10.2.103 JOSCR, Jazelle OS Control Register

The JOSCR characteristics are:

Purpose

A Jazelle register, which provides operating system control of the Jazelle Extension.

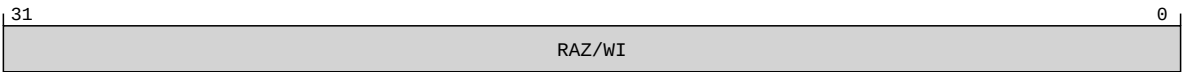
Configuration

There are no configuration notes.

Attributes

JOSCR is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RAZ/WI.

Accessing JOSCR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if ImpDefBool("JOSCR UNDEFINED at EL0") then
        Undefined();
    else
        R(t) = JOSCR();
    end;
elsif PSTATE.EL == EL1 then
    R(t) = JOSCR();
elsif PSTATE.EL == EL2 then
    R(t) = JOSCR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b111	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if ImpDefBool("JOSCR UNDEFINED at EL0") then
        Undefined();
    else
        return;
    end;
elsif PSTATE.EL == EL1 then
    return;
elsif PSTATE.EL == EL2 then
    return;
end;
```

D10.2.104 MAIR0, Memory Attribute Indirection Register 0

The MAIR0 characteristics are:

Purpose

Along with MAIR1, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

AttrIndx[2] indicates the MAIR register to be used:

- When AttrIndx[2] is 0, MAIR0 is used.
- When AttrIndx[2] is 1, MAIR1 is used.

Configuration

There are no configuration notes.

Attributes

MAIR0 is a 32-bit register.

This register has the following instances:

- MAIR0.

Field descriptions

TTBCR.EAE=1

31	24	23	16	15	8	7	0
Attr3				Attr2			
Attr1				Attr0			

Attr<n>, bits [8n+7:8n], for n = 3 to 0

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
0b0000	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
0b00RW, RW not 0b00	Normal memory, Outer Write-Through Transient.
0b0100	Normal memory, Outer Non-cacheable.
0b01RW, RW not 0b00	Normal memory, Outer Write-Back Transient.
0b10RW	Normal memory, Outer Write-Through Non-transient.
0b11RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0b0000	Meaning when Attr<n>[7:4] is not 0b0000
0b0000	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
0b0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
0b1000	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=0b00)
0b10RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
0b1100	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=0b00)
0b11RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = MAIR0();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MAIR0();
end;

```


MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        MAIR0() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    MAIR0() = R(t);
end;
```

D10.2.105 MAIR1, Memory Attribute Indirection Register 1

The MAIR1 characteristics are:

Purpose

Along with MAIR0, provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations.

AttrIndx[2] indicates the MAIR register to be used:

- When AttrIndx[2] is 0, MAIR0 is used.
- When AttrIndx[2] is 1, MAIR1 is used.

Configuration

There are no configuration notes.

Attributes

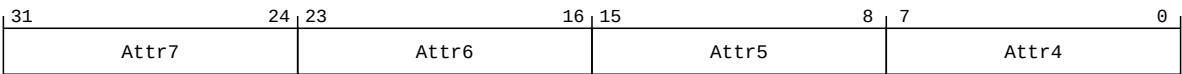
MAIR1 is a 32-bit register.

This register has the following instances:

- MAIR1.

Field descriptions

TTBCR.EAE=1



Attr<n> , bits [8(n-4)+7:8(n-4)], for n = 7 to 4

The memory attribute encoding for an AttrIndx[2:0] entry in a Long descriptor format translation table entry, where:

- AttrIndx[2:0] gives the value of <n> in Attr<n>.
- AttrIndx[2] defines which MAIR to access. Attr7 to Attr4 are in MAIR1, and Attr3 to Attr0 are in MAIR0.

Bits [7:4] are encoded as follows:

Attr<n>[7:4]	Meaning
0b0000	Device memory. See encoding of Attr<n>[3:0] for the type of Device memory.
0b00RW, RW not 0b00	Normal memory, Outer Write-Through Transient.
0b0100	Normal memory, Outer Non-cacheable.
0b01RW, RW not 0b00	Normal memory, Outer Write-Back Transient.
0b10RW	Normal memory, Outer Write-Through Non-transient.
0b11RW	Normal memory, Outer Write-Back Non-transient.

R = Outer Read-Allocate policy, W = Outer Write-Allocate policy.

The meaning of bits [3:0] depends on the value of bits [7:4]:

Attr<n>[3:0]	Meaning when Attr<n>[7:4] is 0b0000	Meaning when Attr<n>[7:4] is not 0b0000
0b0000	Device-nGnRnE memory	UNPREDICTABLE
0b00RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Transient
0b0100	Device-nGnRE memory	Normal memory, Inner Non-cacheable
0b01RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Transient
0b1000	Device-nGRE memory	Normal memory, Inner Write-Through Non-transient (RW=0b00)
0b10RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Through Non-transient
0b1100	Device-GRE memory	Normal memory, Inner Write-Back Non-transient (RW=0b00)
0b11RW, RW not 0b00	UNPREDICTABLE	Normal memory, Inner Write-Back Non-transient

R = Inner Read-Allocate policy, W = Inner Write-Allocate policy.

The R and W bits in some Attr<n> fields have the following meanings:

R or W	Meaning
0b0	No Allocate
0b1	Allocate

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = MAIR1();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MAIR1();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1010	0b0010	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        MAIR1() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    MAIR1() = R(t);
end;
```

D10.2.106 MIDR, Main ID Register

The MIDR characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configuration

Some fields of the MIDR are IMPLEMENTATION DEFINED. For more information about the values of these fields for a particular Armv8 implementation, and any implementation-specific significance of these values, see the product documentation.

AArch32 System register MIDR bits [31:0] are architecturally mapped to External register [MIDR_EL1](#)[31:0].

Attributes

MIDR is a 32-bit register.

Field descriptions

31	24	23	20	19	16	15		4	3	0
Implementer			Variant		Architecture		PartNum			Revision

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Access to this field is RO.

Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() then
        R(t) = VPIDR();
    else
        R(t) = MIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MIDR();
end;
```

D10.2.107 MPIDR, Multiprocessor Affinity Register

The MPIDR characteristics are:

Purpose

In a multiprocessor system, provides an additional PE identification mechanism.

Configuration

In a uniprocessor system, Arm recommends that each Aff<n> field of this register returns a value of 0.

Attributes

MPIDR is a 32-bit register.

Field descriptions

31	30	29	25	24	23	16	15	8	7	0
M	U	RES0			MT	Aff2		Aff1		Aff0

M, bit [31]

Indicates whether this implementation includes the functionality introduced by the Armv7 Multiprocessing Extensions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

M	Meaning
0b0	This implementation does not include the Armv7 Multiprocessing Extensions functionality.
0b1	This implementation includes the Armv7 Multiprocessing Extensions functionality.

Access to this field is RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MPIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() then
        R(t) = VMPIDR();
    else
        R(t) = MPIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MPIDR();
end;
```

D10.2.108 MPUIR, MPU Type Register (EL1)

The MPUIR characteristics are:

Purpose

Identifies the number of regions supported by the EL1 MPU.

Configuration

There are no configuration notes.

Attributes

MPUIR is a 32-bit register.

Field descriptions

31	16	15	8	7	0
RES0		REGION		RES0	

Bits [31:16]

Reserved, RES0.

REGION, bits [15:8]

The number of EL1 MPU regions supported.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [7:0]

Reserved, RES0.

Accessing MPUIR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = MPUIR();
    end;
end;

```

```
elseif PSTATE.EL == EL2 then
    R(t) = MPUIR();
end;
```

D10.2.109 MVFR0, Media and VFP Feature Register 0

The MVFR0 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with [MVFR1](#) and [MVFR2](#).

For general information about the interpretation of the ID registers see ‘[Principles of the ID scheme for fields in ID registers](#)’.

Configuration

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

Attributes

MVFR0 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
FPRound				FPShVec				FPSqrt				FPDivide				FPTrap				FPDP				FPSP				SIMDReg			

FPSHVec	Meaning
0b0000	Short vectors not supported.
0b0001	Short vector operation supported.

All other values are reserved.

In Armv8, the only permitted value is 0b0000.

Access to this field is RO.

FPSqrt, bits [23:20]

Square Root. Indicates whether the floating-point implementation provides support for the ARMv6 VFP square root operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPSqrt	Meaning
0b0000	Not supported in hardware.
0b0001	Supported.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

The VSQRT.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VSQRT.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

Access to this field is RO.

FPDivide, bits [19:16]

Indicates whether the floating-point implementation provides support for VFP divide operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDivide	Meaning
0b0000	Not supported in hardware.
0b0001	Supported.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

The VDIV.F32 instruction also requires the single-precision floating-point attribute, bits [7:4], and the VDIV.F64 instruction also requires the double-precision floating-point attribute, bits [11:8].

Access to this field is RO.

FPTrap, bits [15:12]

Floating-Point Exception Trapping. Indicates whether the floating-point implementation provides support for exception trapping.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPTrip	Meaning
0b0000	Not supported.
0b0001	Supported.

All other values are reserved.

A value of 0b0001 indicates that, when the corresponding trap is enabled, a floating-point exception generates an exception.

Access to this field is RO.

FPDP, bits [11:8]

Floating-Point Double-Precision. Indicates whether the floating-point implementation provides support for double-precision operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDP	Meaning
0b0000	Not supported in hardware.
0b0001	Supported, VFPv2.
0b0010	Supported, VFPv3, VFPv4, or Armv8. VFPv3 and Armv8 add an instruction to load a double-precision floating-point constant, and conversions between double-precision and fixed-point values.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0010.

A value of 0b0001 or 0b0010 indicates support for all VFP double-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F64 is only available if the Square root field is 0b0001.
- VDIV.F64 is only available if the Divide field is 0b0001.
- Conversion between double-precision and single-precision is only available if the single-precision field is nonzero.

Access to this field is RO.

FPSP, bits [7:4]

Floating-Point Single-Precision. Indicates whether the floating-point implementation provides support for single-precision operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPSP	Meaning
0b0000	Not supported in hardware.

FPSP	Meaning
0b0001	Supported, VFPv2.
0b0010	Supported, VFPv3 or VFPv4. VFPv3 adds an instruction to load a single-precision floating-point constant, and conversions between single-precision and fixed-point values.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0010.

A value of 0b0001 or 0b0010 indicates support for all VFP single-precision instructions in the supported version of VFP, except that, in addition to this field being nonzero:

- VSQRT.F32 is only available if the Square root field is 0b0001.
- VDIV.F32 is only available if the Divide field is 0b0001.
- Conversion between double-precision and single-precision is only available if the double-precision field is nonzero.

Access to this field is RO.

SIMDReg, bits [3:0]

Advanced SIMD registers. Indicates whether the Advanced SIMD and floating-point implementation provides support for the Advanced SIMD and floating-point register bank.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDReg	Meaning
0b0000	The implementation has no Advanced SIMD and floating-point support.
0b0001	The implementation includes floating-point support with 16 x 64-bit registers.
0b0010	The implementation includes Advanced SIMD and floating-point support with 32 x 64-bit registers.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0010.

In Armv8-R AArch32, the value 0b0001 is also permitted.

Access to this field is RO.

Accessing MVFR0

Accesses to this register use the following encodings in the System register encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg

0b0111

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
```



```
        Undefined();
    elseif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    elseif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        R(t) = MVFR0();
    end;
elseif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        R(t) = MVFR0();
    end;
end;
```

D10.2.110 MVFR1, Media and VFP Feature Register 1

The MVFR1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with [MVFR0](#) and [MVFR2](#).

For general information about the interpretation of the ID registers see ‘[Principles of the ID scheme for fields in ID registers](#)’.

Configuration

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

Attributes

MVFR1 is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
SIMDFMAC				FPHP				SIMDHP				SIMDSP			
								SIMDInt				SIMDLS			
												FPDNaN			
												FPFtZ			

SIMDFMAC, bits [31:28]

Advanced SIMD Fused Multiply-Accumulate. Indicates whether the Advanced SIMD implementation provides fused multiply accumulate instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDFMAC	Meaning
0b0000	Not implemented.
0b0001	Implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

The Advanced SIMD and floating-point implementations must provide the same level of support for these instructions.

Access to this field is RO.

FPHP, bits [27:24]

Floating-Point Half-Precision. Indicates the level of half-precision floating-point support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPHP	Meaning
0b0000	Not supported.
0b0001	Floating-point half-precision conversion instructions are supported for conversion between single-precision and half-precision.
0b0010	As for 0b0001, and adds instructions for conversion between double-precision and half-precision.
0b0011	As for 0b0010, and adds support for half-precision floating-point arithmetic.

The table above shows all possible values of this field. The values this field can take in the current implementation are:

- 0b0000
- 0b0001
- 0b0010

All other values are reserved.

In Armv8-R AArch32, the permitted values are 0b0000, 0b0001 and 0b0010.

The level of support indicated by this field must be equivalent to the level of support indicated by the SIMDHP field, meaning the permitted values are:

Half-Precision instructions supported	FPHP	SIMDHP
No support	0b0000	0b0000
Conversions only	0b0010	0b0001

Access to this field is RO.

SIMDHP, bits [23:20]

Advanced SIMD Half-Precision. Indicates the level of half-precision floating-point support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDHP	Meaning
0b0000	Not supported.
0b0001	SIMD half-precision conversion instructions are supported for conversion between single-precision and half-precision.
0b0010	As for 0b0001, and adds support for half-precision floating-point arithmetic.

The table above shows all possible values of this field. The values this field can take in the current implementation are:

- 0b0000
- 0b0001

All other values are reserved.

In Armv8-R AArch32, the permitted value are 0b0000 and 0b0001.

The level of support indicated by this field must be equivalent to the level of support indicated by the FPHP field, meaning the permitted values are:

Half-Precision instructions supported	FPHP	SIMDHP
No support	0b0000	0b0000
Conversions only	0b0010	0b0001

Access to this field is RO.

SIMDSP, bits [19:16]

Advanced SIMD Single-Precision. Indicates whether the Advanced SIMD and floating-point implementation provides single-precision floating-point instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDSP	Meaning
0b0000	Not implemented.
0b0001	Implemented. This value is permitted only if the SIMDInt field is 0b0001.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SIMDInt, bits [15:12]

Advanced SIMD Integer. Indicates whether the Advanced SIMD and floating-point implementation provides integer instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDInt	Meaning
0b0000	Not implemented.
0b0001	Implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SIMDLS, bits [11:8]

Advanced SIMD Load/Store. Indicates whether the Advanced SIMD and floating-point implementation provides load/store instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDLS	Meaning
0b0000	Not implemented.
0b0001	Implemented.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

FPDNaN, bits [7:4]

Default NaN. Indicates whether the floating-point implementation provides support only for the default NaN.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPDNaN	Meaning
0b0000	Not implemented, or hardware supports only the default NaN.
0b0001	Hardware supports propagation of NaN values.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

FPFtZ, bits [3:0]

Flush to Zero mode. Indicates whether the floating-point implementation provides support only for the Flush-to-Zero mode of operation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPFtZ	Meaning
0b0000	Not implemented, or hardware supports only the Flush-to-Zero mode of operation.
0b0001	Hardware supports full denormalized number arithmetic.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Accessing MVFR1

Accesses to this register use the following encodings in the System register encoding space:

VMRS{<c>}{<q>} <Rt>, <spec_reg>

reg
0b0110

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
        Undefined();
    elseif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    elseif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        R(t) = MVFR1();
    end;
elseif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        R(t) = MVFR1();
    end;
end;
```

D10.2.111 MVFR2, Media and VFP Feature Register 2

The MVFR2 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with [MVFR0](#) and [MVFR1](#).

For general information about the interpretation of the ID registers see ‘[Principles of the ID scheme for fields in ID registers](#)’.

Configuration

Implemented only if the implementation includes Advanced SIMD and floating-point instructions.

Attributes

MVFR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

FPMisc, bits [7:4]

Indicates whether the floating-point implementation provides support for miscellaneous VFP features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FPMisc	Meaning
0b0000	Not implemented, or no support for miscellaneous features.
0b0001	Support for Floating-point selection.
0b0010	As 0b0001, and Floating-point Conversion to Integer with Directed Rounding modes.
0b0011	As 0b0010, and Floating-point Round to Integer Floating-point.
0b0100	As 0b0011, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0100.

Access to this field is RO.

SIMDMisc, bits [3:0]

Indicates whether the Advanced SIMD implementation provides support for miscellaneous Advanced SIMD features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIMDMisc	Meaning
0b0000	Not implemented, or no support for miscellaneous features.
0b0001	Floating-point Conversion to Integer with Directed Rounding modes.
0b0010	As 0b0001, and Floating-point Round to Integer Floating-point.
0b0011	As 0b0010, and Floating-point MaxNum and MinNum.

All other values are reserved.

In Armv8, the permitted values are 0b0000 and 0b0011.

Access to this field is RO.

Accessing MVFR2

Accesses to this register use the following encodings in the System register encoding space:

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

reg
0b0101

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if CPACR().cp10 == '00' then
        Undefined();
    elsif EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x08);
    elsif EL2Enabled() && HCR().TID3 == '1' then
        AArch32_TakeHypTrapException(0x08);
    else
        R(t) = MVFR2();
    end;
elsif PSTATE.EL == EL2 then
    if EL2Enabled() && HCPTR().TCP10 == '1' then
        AArch32_TakeHypTrapException(0x00);
    else
        R(t) = MVFR2();
    end;
end;
```


D10.2.112 NSACR, Non-Secure Access Control Register

The NSACR characteristics are:

Purpose

Defines the access permissions to Trace, Advanced SIMD and floating-point functionality. Also includes IMPLEMENTATION DEFINED bits that can define Non-secure access permissions for IMPLEMENTATION DEFINED functionality.

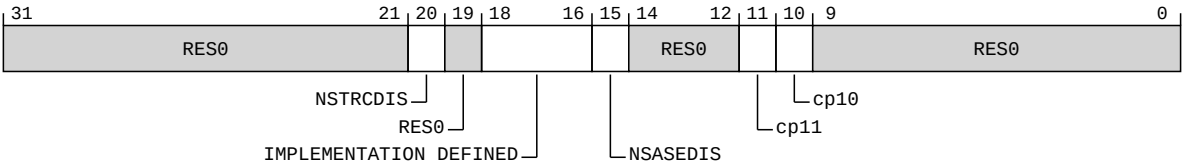
Configuration

Any read of the NSACR from EL2 or EL1 returns a value of 0x00000C00.
Any write accesses, unless trapped to EL2, are UNDEFINED

Attributes

NSACR is a 32-bit register.

Field descriptions



Bits [31:21]

Reserved, RES0.

NSTRCDIS, bit [20]

Disables Non-secure System register accesses to all implemented trace System registers.

NSTRCDIS	Meaning
0b0	This control has no effect on: - System register access to implemented trace System registers. - The behavior of CPACR.TRCDIS and HCPTR.TTA.
0b1	Non-secure System register accesses to all implemented trace System registers are disabled, meaning: - CPACR.TRCDIS behaves as RAO/WI in Non-secure state, regardless of its actual value. - HCPTR.TTA behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.TRCDIS field:

- If CPACR.TRCDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.TRCDIS is RW, this field is RW.

Note

- The ETMv4 architecture does not permit EL0 to access the trace System registers. If the trace unit implements FEAT_ETMv4, EL0 accesses to the trace System registers are UNDEFINED.
- The architecture does not provide Non-secure access controls on trace register accesses through the optional memory-mapped external debug interface.

System register accesses to the trace System registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

Bit [19]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [18:16]

IMPLEMENTATION DEFINED.

NSASEDIS, bit [15]

Disables Non-secure access to the Advanced SIMD functionality.

NSASEDIS	Meaning
0b0	This control has no effect on: • Non-secure access to Advanced SIMD functionality. • The behavior of CPACR.ASEDIS and HCPTR.TASE.
0b1	Non-secure access to the Advanced SIMD functionality is disabled, meaning: • CPACR.ASEDIS behaves as RAO/WI in Non-secure state, regardless of its actual value. • HCPTR.TASE behaves as RAO/WI, regardless of its actual value.

The implementation of this field must correspond to the implementation of the CPACR.ASEDIS field:

- If CPACR.ASEDIS is RES0, this field is RES0. If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.
- If CPACR.ASEDIS is RAZ/WI, this field is RAZ/WI.
- If CPACR.ASEDIS is RW, this field is RW.

Bits [14:12]

Reserved, RES0.

cp11, bit [11]

The value of this field is ignored. If this field is programmed with a different value to the cp10 field then this field is UNKNOWN on a direct read of the NSACR.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

cp10, bit [10]

Enable Non-secure access to the Advanced SIMD and floating-point features. Possible values of the fields are:

cp10	Meaning
0b0	Advanced SIMD and floating-point features can be accessed only from Secure state. Any attempt to access this functionality from Non-secure state is UNDEFINED. When the PE is in Non-secure state: - The CPACR.{cp11, cp10} fields ignore writes and read as 0b00, access denied. - The HCPTR.{TCP11, TCP10} fields behave as RAO/WI, regardless of their actual values.
0b1	Advanced SIMD and floating-point features can be accessed from both Security states.

If Non-secure access to the Advanced SIMD and floating-point functionality is enabled, the CPACR must be checked to determine the level of access that is permitted.

The Advanced SIMD and floating-point features controlled by these fields are:

- Execution of any floating-point or Advanced SIMD instruction.
- Any access to the Advanced SIMD and floating-point registers D0-D31 and their views as S0-S31 and Q0-Q15.
- Any access to the FPSCR, FPSID, MVFR0, MVFR1, MVFR2, or FPEXC System registers.

If the implementation does not include Advanced SIMD and floating-point functionality, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing NSACR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b010

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = Zeros{20} :: '1100' :: Zeros{8};
    end;
elsif PSTATE.EL == EL2 then
    R(t) = Zeros{20} :: '1100' :: Zeros{8};
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    Undefined();
end;
```

D10.2.113 PAR, Physical Address Register

The PAR characteristics are:

Purpose

Returns the output address (OA) from an Address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

Configuration

PAR returns a 64-bit value.

Attributes

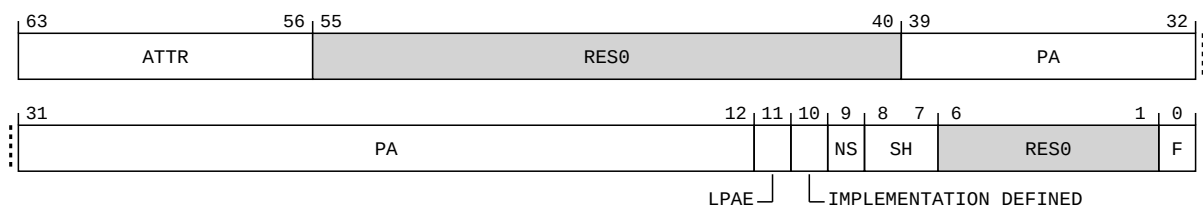
PAR is a 64-bit register.

This register has the following instances:

- PAR.

Field descriptions

When the instruction returned a 64-bit value to the PAR, PAR.F==0:



This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR can return a value that indicates the resulting attributes. More precisely:

- Memory attribute fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits. This applies to the ATTR and SH fields.
- See the NS bit description for constraints on the value it returns.

ATTR, bits [63:56]

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR0 and MAIR1.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [55:40]

Reserved, RES0.

PA, bits [39:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[39:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

LPAE	Meaning
0b1	PAR returned a 64-bit value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bit [10]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [9]

In Armv8-R AArch32, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is UNKNOWN/WI.

SH, bits [8:7]

Shareability attribute, for the returned output address. Permitted values are:

SH	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

The value 0b01 is reserved.

Note

This field returns the value 0b10 for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:1]

Reserved, RES0.

F, bit [0]

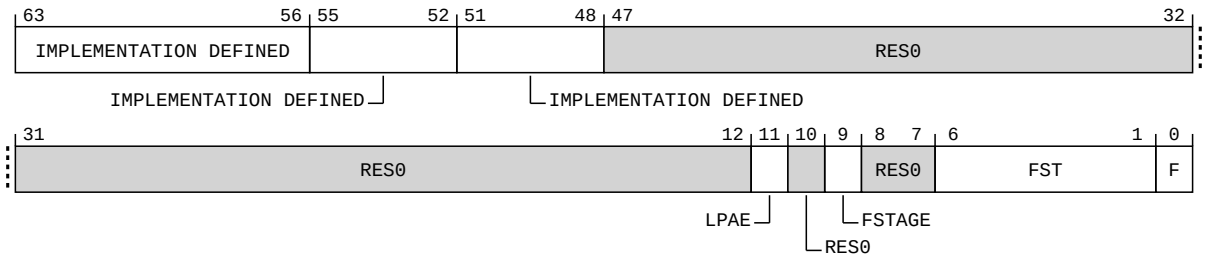
Indicates whether the instruction performed a successful address translation.

F	Meaning
0b0	Address translation completed successfully.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When the instruction returned a 64-bit value to the PAR, PAR.F==1:



This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

IMPLEMENTATION DEFINED, bits [63:56]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [47:12]

Reserved, RES0.

LPAE, bit [11]

When updating the PAR with the result of the translation operation, this bit is set as follows:

LPAE	Meaning
0b1	PAR returned a 64-bit value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [10]

Reserved, RES0.

FSTAGE, bit [9]

Indicates the translation stage at which the translation aborted:

FSTAGE	Meaning
0b0	Translation aborted because of a fault in the stage 1 translation.
0b1	Translation aborted because of a fault in the stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:7]

Reserved, RES0.

FST, bits [6:1]

Fault status field. Values are as in the [DFSR.STATUS](#) and [IFSR.STATUS](#) fields.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
0b1	Address translation aborted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0100	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PAR()[31:0];
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PAR()[31:0];
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0111	0b0100	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PAR()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PAR()[31:0] = R(t);
end;

```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0111	0b0000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x04);
    else

```

```
        R(t, t2) = PAR();
    end;
elseif PSTATE.EL == EL2 then
    R(t, t2) = PAR();
end;
```

MCCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b0111	0b0000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T7 == '1' then
        AArch32_TakeHypTrapException(0x04);
    else
        PAR() = R(t, t2);
    end;
elseif PSTATE.EL == EL2 then
    PAR() = R(t, t2);
end;
```

D10.2.114 PRBAR, Protection Region Base Address Register (EL1)

The PRBAR characteristics are:

Purpose

Provides access to the base addresses for the EL1 MPU region. **PRSELR.REGION** determines which MPU region is selected.

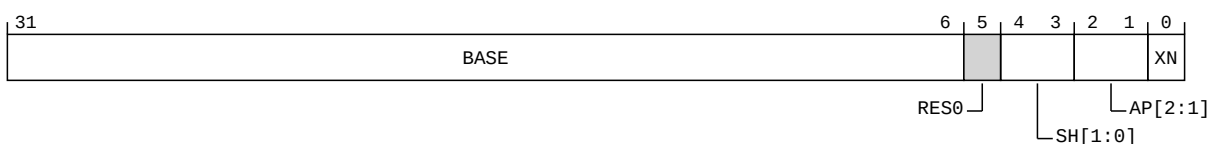
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRBAR is a 32-bit register.

Field descriptions

**BASE, bits [31:6]**

Bits[31:6] of the lower inclusive limit of the selected EL1 MPU memory region. This value is zero extended to provide the base address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SH[1:0], bits [4:3]

Shareability attribute SH[1:0].

SH[1:0]	Meaning
0b00	Non-shareable
0b01	Reserved, CONSTRAINED UNPREDICTABLE
0b10	Outer Shareable
0b11	Inner Shareable

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AP[2:1], bits [2:1]

Access Permission attributes AP[2:1].

AP[2:1]	Meaning
0b00	Read/write at EL1, no access at EL0
0b01	Read/write at EL1 and EL0
0b10	Read-only at EL1, no access at EL0
0b11	Read-only at EL1 and EL0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

XN, bit [0]

Execute Never

XN	Meaning
0b0	Execution of instructions fetched from the region is permitted.
0b1	Execution of instructions fetched from the region is not permitted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PRBAR

Any access to MPU region register PRBAR above the number of implemented regions specified by MPUIR.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRBAR register return an UNKNOWN value.
- Writes to unimplemented PRBAR register make all PRBAR registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0011	0b000

```

if UInt(PRSELR().REGION) >= UInt(MPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);

```

```
        else
            R(t) = PRBAR(UInt(PRSELR().REGION));
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = PRBAR(UInt(PRSELR().REGION));
    end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0011	0b000

```
if UInt(PRSELR().REGION) >= UInt(MPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PRBAR(UInt(PRSELR().REGION)) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PRBAR(UInt(PRSELR().REGION)) = R(t);
end;
```

D10.2.115 PRBAR<n>, Protection Region Base Address Register n (EL1), n = 0 - 31

The PRBAR<n> characteristics are:

Purpose

Provides access to the base address for the EL1 MPU region determined by the value of ‘n’.

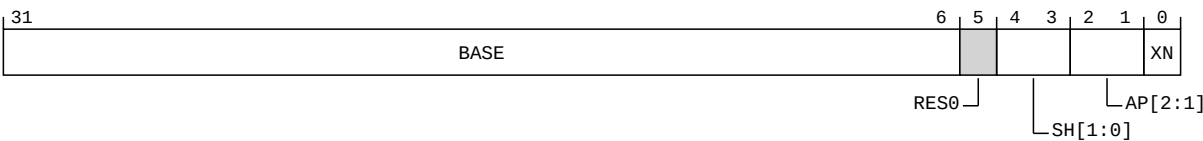
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRBAR<n> is a 32-bit register.

Field descriptions



BASE, bits [31:6]

Bits[31:6] of the lower inclusive limit of the selected EL1 MPU memory region. This value is zero extended to provide the base address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

SH[1:0], bits [4:3]

Shareability attribute SH[1:0].

SH[1:0]	Meaning
0b00	Non-shareable
0b01	Reserved, CONSTRAINED UNPREDICTABLE
0b10	Outer Shareable
0b11	Inner Shareable

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AP[2:1], bits [2:1]

Access Permission attributes AP[2:1].

AP[2:1]	Meaning
0b00	Read/write at EL1, no access at EL0
0b01	Read/write at EL1 and EL0
0b10	Read-only at EL1, no access at EL0
0b11	Read-only at EL1 and EL0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

XN, bit [0]

Execute Never

XN	Meaning
0b0	Execution of instructions fetched from the region is permitted.
0b1	Execution of instructions fetched from the region is not permitted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PRBAR<n>

Any access to MPU region register PRBAR<n> above the number of implemented regions specified by [MPUIR.REGION](#) is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRBAR<n> return an UNKNOWN value.
- Writes to unimplemented PRBAR<n> register make all PRBAR registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where n = 0-31

coproc	opc1	CRn	CRm	opc2
0b1111	0b00:n[4]	0b0110	0b1:n[3:1]	n[0]:0b00

```
let n:integer = UInt(opc1[0] :: CRm[2:0] :: opc2[2]);

if n >= UInt(MPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
```

```
        elsif EL2Enabled() && HCR().TRVM == '1' then
            AArch32_TakeHypTrapException(0x03);
        else
            R(t) = PRBAR(n);
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = PRBAR(n);
    end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where n = 0-31

coproc	opc1	CRn	CRm	opc2
0b1111	0b00:n[4]	0b0110	0b1:n[3:1]	n[0]:0b00

```
let n:integer = UInt(opc1[0] :: CRm[2:0] :: opc2[2]);

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PRBAR(n) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PRBAR(n) = R(t);
end;
```


D10.2.116 PRLAR, Protection Region Limit Address Register (EL1)

The PRLAR characteristics are:

Purpose

Provides access to the limit addresses for the EL1 MPU region. PRSEL.RREGION determines which MPU region is selected.

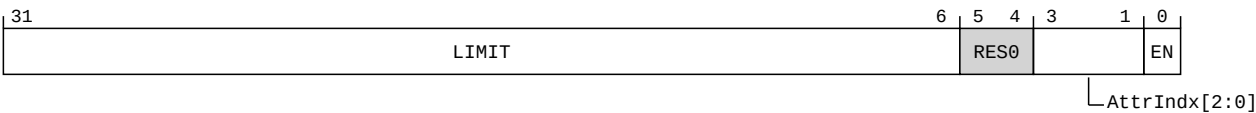
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRLAR is a 32-bit register.

Field descriptions



LIMIT, bits [31:6]

Bits[31:6] of the upper inclusive limit of the selected EL1 MPU memory region. This value is concatenated with the value 0x3F to provide the limit address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [5:4]

Reserved, RES0.

AttrIndx[2:0], bits [3:1]

AttrIndx[2:0] selects attributes from within the associated Memory Attribute Indirection Register.

AttrIndx[2:0]	Meaning
0b000	Select the Attr0 field from MAIR0.
0b001	Select the Attr1 field from MAIR0.
0b010	Select the Attr2 field from MAIR0.
0b011	Select the Attr3 field from MAIR0.
0b100	Select the Attr4 field from MAIR1.
0b101	Select the Attr5 field from MAIR1.

AttrIdx[2:0]	Meaning
0b110	Select the Attr6 field from MAIR1.
0b111	Select the Attr7 field from MAIR1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

EN	Meaning
0b0	Region disabled.
0b1	Region enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PRLAR

Any access to MPU region register PRLAR above the number of implemented regions specified by MPUIR.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRLAR register return an UNKNOWN value.
- Writes to unimplemented PRLAR register make all PRLAR registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0011	0b001

```

if UInt(PRSELR().REGION) >= UInt(MPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PRLAR(UInt(PRSELR().REGION));
    end;
elsif PSTATE.EL == EL2 then

```

```
R(t) = PRLAR(UInt(PRSELR().REGION));
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0011	0b001

```
if UInt(PRSELR().REGION) >= UInt(MPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURRegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PRLAR(UInt(PRSELR().REGION)) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PRLAR(UInt(PRSELR().REGION)) = R(t);
end;
```

D10.2.117 PRLAR<n>, Protection Region Limit Address Register n (EL1), n = 0 - 31

The PRLAR<n> characteristics are:

Purpose

Provides access to the limit address for the EL1 MPU region determined by the value of ‘n’.

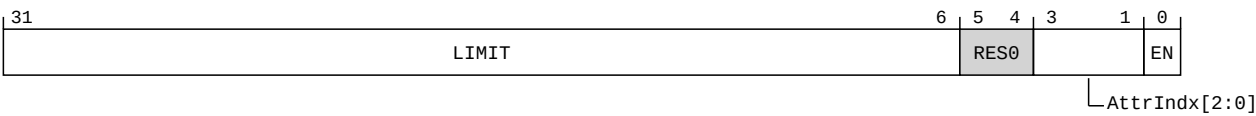
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRLAR<n> is a 32-bit register.

Field descriptions



LIMIT, bits [31:6]

Bits[31:6] of the upper inclusive limit of the selected EL1 MPU memory region. This value is concatenated with the value 0x3F to provide the limit address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [5:4]

Reserved, RES0.

AttrIdx[2:0], bits [3:1]

AttrIdx[2:0] selects attributes from within the associated Memory Attribute Indirection Register.

AttrIdx[2:0]	Meaning
0b000	Select the Attr0 field from MAIR0.
0b001	Select the Attr1 field from MAIR0.
0b010	Select the Attr2 field from MAIR0.
0b011	Select the Attr3 field from MAIR0.
0b100	Select the Attr4 field from MAIR1.
0b101	Select the Attr5 field from MAIR1.

AttrIdx[2:0]	Meaning
0b110	Select the Attr6 field from MAIR1.
0b111	Select the Attr7 field from MAIR1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

EN	Meaning
0b0	Region disabled.
0b1	Region enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PRLAR<n>

Any access to MPU region register PRLAR<n> above the number of implemented regions specified by [MPUIR.REGION](#) is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRLAR<n> return an UNKNOWN value.
- Writes to unimplemented PRLAR<n> register make all PRLAR registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where n = 0-31

coproc	opc1	CRn	CRm	opc2
0b1111	0b00:n[4]	0b0110	0b1:n[3:1]	n[0]:0b01

```
let n:integer = UInt(opc1[0] :: CRm[2:0] :: opc2[2]);

if n >= UInt(MPUIR().REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T10 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PRLAR(n);
```

```
end;  
elseif PSTATE.EL == EL2 then  
    R(t) = PRLAR(n);  
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where n = 0-31

coproc	opc1	CRn	CRm	opc2
0b1111	0b00:n[4]	0b0110	0b1:n[3:1]	n[0]:0b01

```
let n:integer = UInt(opc1[0] :: CRm[2:0] :: opc2[2]);  
  
if n >= UInt(MPUIR().REGION) then  
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);  
elseif PSTATE.EL == EL0 then  
    Undefined();  
elseif PSTATE.EL == EL1 then  
    if EL2Enabled() && HSTR().T10 == '1' then  
        AArch32_TakeHypTrapException(0x03);  
    elseif EL2Enabled() && HCR().TVM == '1' then  
        AArch32_TakeHypTrapException(0x03);  
    else  
        PRLAR(n) = R(t);  
    end;  
elseif PSTATE.EL == EL2 then  
    PRLAR(n) = R(t);  
end;
```

D10.2.118 PRSELR, Protection Region Selection Register (EL1)

The PRSELR characteristics are:

Purpose

Selects the region number for the EL1 MPU region associated with the [PRBAR](#) and [PRLAR](#) registers.

Configuration

There are no configuration notes.

Attributes

PRSELR is a 32-bit register.

Field descriptions

31	8	7	0
RES0			REGION

Bits [31:8]

Reserved, RES0.

REGION, bits [7:0]

The number of the current EL1 MPU region visible in [PRBAR](#) and [PRLAR](#). For N implemented MPU regions, memory region numbering starts at 0 and increments by 1 to the value N-1.

Writing a value greater than or equal to the number of implemented MPU regions specified by [MPUIR](#).REGION, results in CONSTRAINED UNPREDICTABLE behavior.

CONSTRAINED UNPREDICTABLE behavior is that PRSELR register becomes UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PRSELR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0010	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else

```

```
        R(t) = PRSELR();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = PRSELR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T6 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PRSELR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    PRSELR() = R(t);
end;
```


D10.2.119 REVIDR, Revision ID Register

The REVIDR characteristics are:

Purpose

Provides implementation-specific minor revision information.

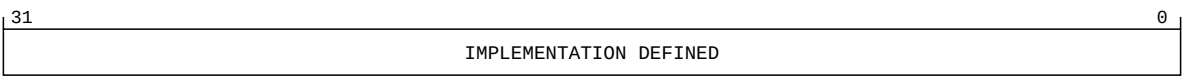
Configuration

If REVIDR has the same value as MIDR, then its contents have no significance.

Attributes

REVIDR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing REVIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b110

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = REVIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = REVIDR();
end;
```

D10.2.120 RVBAR, Reset Vector Base Address Register

The RVBAR characteristics are:

Purpose

This register contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch32 state.

Configuration

There are no configuration notes.

Attributes

RVBAR is a 32-bit register.

Field descriptions



ResetAddress, bits [31:1]

Bits [31:1] of the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 32-bit state.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [0]

Reserved, RES1.

Accessing RVBAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T12 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = RVBAR();
end;
```

D10.2.121 SCTLR, System Control Register

The SCTLR characteristics are:

Purpose

Provides the top-level control of the system, including its memory system.

Configuration

Some bits in the register are read-only. These bits relate to non-configurable features of an implementation, and are provided for compatibility with previous versions of the architecture.

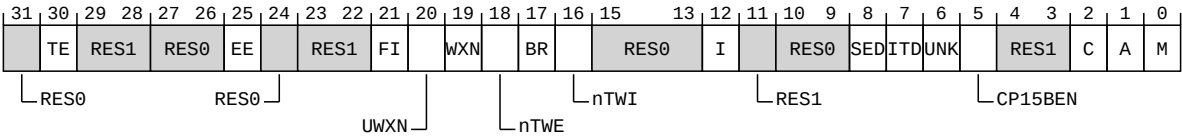
Attributes

SCTLR is a 32-bit register.

This register has the following instances:

- SCTLR.

Field descriptions



Bit [31]

Reserved, RES0.

TE, bit [30]

T32 Exception Enable. This bit controls whether exceptions to an Exception level that is executing at PL1 are taken to A32 or T32 state:

TE	Meaning
0b0	Exceptions, including reset, taken to A32 state.
0b1	Exceptions, including reset, taken to T32 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bits [29:28]

Reserved, RES1.

Bits [27:26]

Reserved, RES0.

EE, bit [25]

The value of the PSTATE.E bit on branch to an exception vector or coming out of reset.

EE	Meaning
0b0	Little-endian. PSTATE.E is cleared to 0 on taking an exception or coming out of reset.
0b1	Big-endian. PSTATE.E is set to 1 on taking an exception or coming out of reset.

If an implementation does not provide Big-endian support for data accesses at Exception levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support for data accesses at Exception levels higher than EL0, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bit [24]

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

FI, bit [21]

Fast Interrupts enable. This bit is a read-only copy of the [HSCTLR.FI](#) bit.

UWXN, bit [20]

Unprivileged write permission implies PL1 XN (Execute-never). This bit can force all memory regions that are writable at PL0 to be treated as XN for accesses from software executing at PL1.

UWXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	Any region that is writable at PL0 forced to XN for accesses from software executing at PL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

WXN, bit [19]

Write permission implies XN (Execute-never). For the PL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN.

WXN	Meaning
0b0	This control has no effect on memory access permissions.

WXN	Meaning
0b1	Any region that is writable in the PL1&0 translation regime is forced to XN for accesses from software executing at PL1 or PL0.

This bit applies only when SCTLR.M bit is set.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to Undefined mode.

nTWE	Meaning
0b0	Any attempt to execute a WFE instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

The attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Note
Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE or WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

BR, bit [17]

Background Region enable. When the EL1 MPU is enabled, this bit controls how an EL1 access that does not map to any MPU memory region is handled.

BR	Meaning
0b0	EL1 MPU Background region disabled. Any EL1 transaction that does not match an EL1 MPU region results in a fault.
0b1	EL1 MPU Background region enabled. For EL1 transactions that do not match an EL1 MPU region, the EL1 Background region attributes are used and access is permitted subject to any stage 2 checks.

This bit only applies to EL1 accesses.

An EL0 access that does not match an EL1 MPU region always results in a Translation fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to Undefined mode.

nTWI	Meaning
0b0	Any attempt to execute a WFI instruction at EL0 is trapped to Undefined mode, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

The attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Note
Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE or WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Bits [15:13]

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL1 and EL0:

I	Meaning
0b0	All instruction access to Normal memory from PL1 and PL0 are Non-cacheable for all levels of instruction and unified cache. If the value of SCTL.R is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
0b1	All instruction access to Normal memory from PL1 and PL0 can be cached at all levels of instruction and unified cache. If the value of SCTL.R is 0, instruction accesses from stage 1 of the PL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

Instruction accesses to Normal memory from EL1 and EL0 are Cacheable regardless of the value of the SCTL.R bit if the value of HCR.DC is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [11]

Reserved, RES1.

Bits [10:9]

Reserved, RES0.

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at PL0 and PL1.

SED	Meaning
0b0	SETEND instruction execution is enabled at PL0 and PL1.
0b1	SETEND instructions are UNDEFINED at PL0 and PL1.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at PL1 and PL0.

ITD	Meaning
0b0	All IT instruction functionality is enabled at PL1 and PL0.
0b1	Any attempt at PL1 or PL0 to execute any of the following is UNDEFINED: - All encodings of the IT instruction with hw1[3:0]!1000. - All encodings of the subsequent instruction with the following values for hw1: 0b11xxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM. 0b1011xxxxxxxxxxxx: All instructions in 'Miscellaneous 16-bit instructions'. 0b10100xxxxxxxxxxxx: ADD Rd, PC, #imm 0b01001xxxxxxxxxxxx: LDR Rd, [PC, #imm] 0b0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC. 0b010001xx1xxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn. These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block. It is IMPLEMENTATION DEFINED whether the IT instruction is treated as: - A 16-bit instruction, that can only be followed by another 16-bit instruction. - The first half of a 32-bit instruction. This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED. An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1, then behavior is CONSTRAINED UNPREDICTABLE. For more information see 'Changes to an ITD control by an instruction in an IT block'.

ITD is optional, but if it is implemented in the SCTLR, then it must also be implemented in the HSCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

When an implementation does not implement ITD, access to this field is RAZ/WI.

UNK, bit [6]

Writes to this bit are IGNORED. Reads of this bit return an UNKNOWN value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from PL1 and PL0:

CP15BEN	Meaning
0b0	PL0 and PL1 execution of the CP15DMB , CP15DSB , and CP15ISB instructions is UNDEFINED.
0b1	PL0 and PL1 execution of the CP15DMB , CP15DSB , and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR, then it must also be implemented in the [HSCTLR](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

When an implementation does not implement CP15BEN, access to this field is RAO/WI.

Bits [4:3]

Reserved, RES1.

C, bit [2]

Cacheability control, for data accesses at EL1 and EL0:

C	Meaning
0b0	All data access to Normal memory from PL1 and PL0 are Non-cacheable for all levels of data and unified cache.
0b1	All data access to Normal memory from PL1 and PL0 can be cached at all levels of data and unified cache.

The PE ignores SCTLR.C, and data accesses to Normal memory from EL1 and EL0 are Cacheable, if the value of [HCR.DC](#) is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at PL1 and PL0:

A	Meaning
0b0	Alignment fault checking disabled when executing at PL1 or PL0. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.
0b1	Alignment fault checking enabled when executing at PL1 or PL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

M, bit [0]

MPU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

M	Meaning
0b0	EL1 and EL0 stage 1 address translation disabled. See the SCTLR.I field for the behavior of instruction accesses to Normal memory.
0b1	EL1 and EL0 stage 1 address translation enabled.

The PE behaves as if the value of the SCTLR.M field is 0 for all purposes other than returning the value of a direct read of the field if HCR.TGE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing SCTLR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TRVM == '1' then
        AArch32_TakeHypTrapException(0x03);

```

```
        else
            R(t) = SCTLRL();
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = SCTLRL();
    end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TVM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        SCTLRL() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    SCTLRL() = R(t);
end;
```

D10.2.122 SPSR, Saved Program Status Register

The SPSR characteristics are:

Purpose

Holds the saved process state for the current mode.

Configuration

This register is a Special-purpose register.

Attributes

SPSR is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	21	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q			J	RES0	IL		GE			IT[7:2]		E	A	I	F	T	M[4:0]

└─ IT[1:0]

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to the current mode, and copied to PSTATE.N on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to the current mode, and copied to PSTATE.Z on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to the current mode, and copied to PSTATE.C on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to the current mode, and copied to PSTATE.V on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to the current mode, and copied to PSTATE.Q on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to the current mode, and copied to PSTATE.IT on executing an exception return operation in the current mode.

On executing an exception return operation in the current mode, SPSR.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Access to this field is RES0.

Bits [23:21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to the current mode, and copied to PSTATE.IL on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to the current mode, and copied to PSTATE.GE on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to the current mode, and copied to PSTATE.E on executing an exception return operation in the current mode.

If the implementation does not support big-endian operation, SPSR.E is RES0. If the implementation does not support little-endian operation, SPSR.E is RES1. On executing an exception return operation in the current mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to the current mode, and copied to PSTATE.A on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to the current mode, and copied to PSTATE.I on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to the current mode, and copied to PSTATE.F on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to the current mode, and copied to PSTATE.T on executing an exception return operation in the current mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to the current mode, and copied to PSTATE.M[4:0] on executing an exception return operation in the current mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in the current mode is an illegal return event, as described in [‘Illegal return events from AArch32 state’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR

SPSR can be read using the MRS instruction and written using the MSR (register) or MSR (immediate) instructions.

The MSR (register) and MRS instructions used to access SPSR are data-independent-time instructions as described in [About PSTATE.DIT](#).

D10.2.123 SPSR_abt, Saved Program Status Register (Abort mode)

The SPSR_abt characteristics are:

Purpose

Holds the saved process state when an exception is taken to Abort mode.

Configuration

This register is a Special-purpose register.

Attributes

SPSR_abt is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	21	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q			J	RES0	IL		GE		IT[7:2]		E	A	I	F	T		M[4:0]

└─ IT[1:0]

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Abort mode, and copied to PSTATE.N on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Abort mode, and copied to PSTATE.Z on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Abort mode, and copied to PSTATE.C on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Abort mode, and copied to PSTATE.V on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Abort mode, and copied to PSTATE.Q on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Abort mode, and copied to PSTATE.IT on executing an exception return operation in Abort mode.

On executing an exception return operation in Abort mode, SPSR_abt.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Access to this field is RES0.

Bits [23:21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Abort mode, and copied to PSTATE.IL on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Abort mode, and copied to PSTATE.GE on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Abort mode, and copied to PSTATE.E on executing an exception return operation in Abort mode.

If the implementation does not support big-endian operation, SPSR_abt.E is RES0. If the implementation does not support little-endian operation, SPSR_abt.E is RES1. On executing an exception return operation in Abort mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_abt.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_abt.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Abort mode, and copied to PSTATE.A on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Abort mode, and copied to PSTATE.I on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Abort mode, and copied to PSTATE.F on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Abort mode, and copied to PSTATE.T on executing an exception return operation in Abort mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Abort mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Abort mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_abt.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Abort mode is an illegal return event, as described in [‘Illegal return events from AArch32 state’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_abt

SPSR_abt is accessible in all modes other than User mode and Abort mode.

The MSR (register) and MRS instructions used to access SPSR_abt are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

MRS{<c>}{<q>} <Rd>, SPSR_abt

R	M	M1
0b1	0b1	0b0100

MSR{<c>}{<q>} SPSR_abt, <Rn>

R	M	M1
0b1	0b1	0b0100

D10.2.124 SPSR_fiq, Saved Program Status Register (FIQ mode)

The SPSR_fiq characteristics are:

Purpose

Holds the saved process state when an exception is taken to FIQ mode.

Configuration

This register is a Special-purpose register.

Attributes

SPSR_fiq is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	21	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q			J	RES0	IL		GE		IT[7:2]		E	A	I	F	T		M[4:0]

└─ IT[1:0]

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to FIQ mode, and copied to PSTATE.N on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to FIQ mode, and copied to PSTATE.Z on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to FIQ mode, and copied to PSTATE.C on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to FIQ mode, and copied to PSTATE.V on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to FIQ mode, and copied to PSTATE.Q on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to FIQ mode, and copied to PSTATE.IT on executing an exception return operation in FIQ mode.

On executing an exception return operation in FIQ mode, SPSR_fiq.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Access to this field is RES0.

Bits [23:21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to FIQ mode, and copied to PSTATE.IL on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to FIQ mode, and copied to PSTATE.GE on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to FIQ mode, and copied to PSTATE.E on executing an exception return operation in FIQ mode.

If the implementation does not support big-endian operation, SPSR_fiq.E is RES0. If the implementation does not support little-endian operation, SPSR_fiq.E is RES1. On executing an exception return operation in FIQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_fiq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_fiq.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to FIQ mode, and copied to PSTATE.A on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to FIQ mode, and copied to PSTATE.I on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to FIQ mode, and copied to PSTATE.F on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to FIQ mode, and copied to PSTATE.T on executing an exception return operation in FIQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to FIQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in FIQ mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_fiq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in FIQ mode is an illegal return event, as described in [‘Illegal return events from AArch32 state’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_fiq

SPSR_fiq is accessible in all modes other than User mode and FIQ mode.

The MSR (register) and MRS instructions used to access SPSR_fiq are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

MRS{<c>}{<q>} <Rd>, SPSR_fiq

R	M	M1
0b1	0b0	0b1110

MSR{<c>}{<q>} SPSR_fiq, <Rn>

R	M	M1
0b1	0b0	0b1110

D10.2.125 SPSR_hyp, Saved Program Status Register (Hyp mode)

The SPSR_hyp characteristics are:

Purpose

Holds the saved process state when an exception is taken to Hyp mode.

Configuration

This register is a Special-purpose register.

Attributes

SPSR_hyp is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	21	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q			J	RES0	IL		GE		IT[7:2]		E	A	I	F	T		M[4:0]

└─ IT[1:0]

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Hyp mode, and copied to PSTATE.N on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Hyp mode, and copied to PSTATE.Z on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Hyp mode, and copied to PSTATE.C on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Hyp mode, and copied to PSTATE.V on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Hyp mode, and copied to PSTATE.Q on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Hyp mode, and copied to PSTATE.IT on executing an exception return operation in Hyp mode.

On executing an exception return operation in Hyp mode, SPSR_hyp.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Access to this field is RES0.

Bits [23:21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Hyp mode, and copied to PSTATE.IL on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Hyp mode, and copied to PSTATE.GE on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Hyp mode, and copied to PSTATE.E on executing an exception return operation in Hyp mode.

If the implementation does not support big-endian operation, SPSR_hyp.E is RES0. If the implementation does not support little-endian operation, SPSR_hyp.E is RES1. On executing an exception return operation in Hyp mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_hyp.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_hyp.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Hyp mode, and copied to PSTATE.A on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Hyp mode, and copied to PSTATE.I on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Hyp mode, and copied to PSTATE.F on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Hyp mode, and copied to PSTATE.T on executing an exception return operation in Hyp mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Hyp mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Hyp mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_hyp.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Hyp mode is an illegal return event, as described in [‘Illegal return events from AArch32 state’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_hyp

SPSR_hyp is accessible only in Monitor mode.

The MSR (register) and MRS instructions used to access SPSR_hyp are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

MRS{<c>}{<q>} <Rd>, SPSR_hyp

R	M	M1
0b1	0b1	0b1110

MSR{<c>}{<q>} SPSR_hyp, <Rn>

R	M	M1
0b1	0b1	0b1110

D10.2.126 SPSR_irq, Saved Program Status Register (IRQ mode)

The SPSR_irq characteristics are:

Purpose

Holds the saved process state when an exception is taken to IRQ mode.

Configuration

This register is a Special-purpose register.

Attributes

SPSR_irq is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	21	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q			J	RES0	IL		GE		IT[7:2]		E	A	I	F	T		M[4:0]

└─ IT[1:0]

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to IRQ mode, and copied to PSTATE.N on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to IRQ mode, and copied to PSTATE.Z on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to IRQ mode, and copied to PSTATE.C on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to IRQ mode, and copied to PSTATE.V on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to IRQ mode, and copied to PSTATE.Q on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to IRQ mode, and copied to PSTATE.IT on executing an exception return operation in IRQ mode.

On executing an exception return operation in IRQ mode, SPSR_irq.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Access to this field is RES0.

Bits [23:21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to IRQ mode, and copied to PSTATE.IL on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to IRQ mode, and copied to PSTATE.GE on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to IRQ mode, and copied to PSTATE.E on executing an exception return operation in IRQ mode.

If the implementation does not support big-endian operation, SPSR_irq.E is RES0. If the implementation does not support little-endian operation, SPSR_irq.E is RES1. On executing an exception return operation in IRQ mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_irq.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_irq.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to IRQ mode, and copied to PSTATE.A on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to IRQ mode, and copied to PSTATE.I on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to IRQ mode, and copied to PSTATE.F on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to IRQ mode, and copied to PSTATE.T on executing an exception return operation in IRQ mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to IRQ mode, and copied to PSTATE.M[4:0] on executing an exception return operation in IRQ mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_irq.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in IRQ mode is an illegal return event, as described in [‘Illegal return events from AArch32 state’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_irq

SPSR_irq is accessible in all modes other than User mode and IRQ mode.

The MSR (register) and MRS instructions used to access SPSR_irq are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

MRS{<c>}{<q>} <Rd>, SPSR_irq

R	M	M1
0b1	0b1	0b0000

MSR{<c>}{<q>} SPSR_irq, <Rn>

R	M	M1
0b1	0b1	0b0000

D10.2.127 SPSR_svc, Saved Program Status Register (Supervisor mode)

The SPSR_svc characteristics are:

Purpose

Holds the saved process state when an exception is taken to Supervisor mode.

Configuration

This register is a Special-purpose register.

Attributes

SPSR_svc is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	21	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q			J	RES0	IL		GE			IT[7:2]		E	A	I	F	T	M[4:0]

└─ IT[1:0]

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Supervisor mode, and copied to PSTATE.N on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Supervisor mode, and copied to PSTATE.Z on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Supervisor mode, and copied to PSTATE.C on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Supervisor mode, and copied to PSTATE.V on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Supervisor mode, and copied to PSTATE.Q on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Supervisor mode, and copied to PSTATE.IT on executing an exception return operation in Supervisor mode.

On executing an exception return operation in Supervisor mode, SPSR_svc.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Access to this field is RES0.

Bits [23:21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Supervisor mode, and copied to PSTATE.IL on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Supervisor mode, and copied to PSTATE.GE on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Supervisor mode, and copied to PSTATE.E on executing an exception return operation in Supervisor mode.

If the implementation does not support big-endian operation, SPSR_svc.E is RES0. If the implementation does not support little-endian operation, SPSR_svc.E is RES1. On executing an exception return operation in Supervisor mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_svc.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_svc.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Supervisor mode, and copied to PSTATE.A on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Supervisor mode, and copied to PSTATE.I on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Supervisor mode, and copied to PSTATE.F on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Supervisor mode, and copied to PSTATE.T on executing an exception return operation in Supervisor mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Supervisor mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Supervisor mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_svc.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Supervisor mode is an illegal return event, as described in [‘Illegal return events from AArch32 state’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_svc

SPSR_svc is accessible in all modes other than User mode and Supervisor mode.

The MSR (register) and MRS instructions used to access SPSR_svc are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

MRS{<c>}{<q>} <Rd>, SPSR_svc

R	M	M1
0b1	0b1	0b0010

MSR{<c>}{<q>} SPSR_svc, <Rn>

R	M	M1
0b1	0b1	0b0010

D10.2.128 SPSR_und, Saved Program Status Register (Undefined mode)

The SPSR_und characteristics are:

Purpose

Holds the saved process state when an exception is taken to Undefined mode.

Configuration

This register is a Special-purpose register.

Attributes

SPSR_und is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	21	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q			J	RES0	IL		GE			IT[7:2]		E	A	I	F	T	M[4:0]

└─ IT[1:0]

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to Undefined mode, and copied to PSTATE.N on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to Undefined mode, and copied to PSTATE.Z on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to Undefined mode, and copied to PSTATE.C on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to Undefined mode, and copied to PSTATE.V on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to Undefined mode, and copied to PSTATE.Q on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on taking an exception to Undefined mode, and copied to PSTATE.IT on executing an exception return operation in Undefined mode.

On executing an exception return operation in Undefined mode, SPSR_und.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

J, bit [24]

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state.

Armv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Access to this field is RES0.

Bits [23:21]

Reserved, RES0.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to Undefined mode, and copied to PSTATE.IL on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to Undefined mode, and copied to PSTATE.GE on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to Undefined mode, and copied to PSTATE.E on executing an exception return operation in Undefined mode.

If the implementation does not support big-endian operation, SPSR_und.E is RES0. If the implementation does not support little-endian operation, SPSR_und.E is RES1. On executing an exception return operation in Undefined mode, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_und.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_und.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to Undefined mode, and copied to PSTATE.A on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to Undefined mode, and copied to PSTATE.I on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to Undefined mode, and copied to PSTATE.F on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to Undefined mode, and copied to PSTATE.T on executing an exception return operation in Undefined mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on taking an exception to Undefined mode, and copied to PSTATE.M[4:0] on executing an exception return operation in Undefined mode.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If SPSR_und.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in Undefined mode is an illegal return event, as described in [‘Illegal return events from AArch32 state’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_und

SPSR_und is accessible in all modes other than User mode and Undefined mode.

The MSR (register) and MRS instructions used to access SPSR_und are data-independent-time instructions as described in About PSTATE.DIT.

Accesses to this register use the following encodings in the System register encoding space:

MRS{<c>}{<q>} <Rd>, SPSR_und

R	M	M1
0b1	0b1	0b0110

MSR{<c>}{<q>} SPSR_und, <Rn>

R	M	M1
0b1	0b1	0b0110

D10.2.129 TCMTR, TCM Type Register

The TCMTR characteristics are:

Purpose

Provides information about the implementation of the TCM.

Configuration

If EL1 or above can use AArch32 then this register must be implemented.

Attributes

TCMTR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing TCMTR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = TCMTR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = TCMTR();
end;
```

D10.2.130 TLBIALL, TLB Invalidate All

The TLBIALL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
 - Would be required for the EL1&0 translation regime.
 - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the PL1&0 translation regime and matches the current VMID.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIALL is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr);
end;
```

D10.2.131 TLBIALLH, TLB Invalidate All, Hyp mode

The TLBIALLH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIALLH is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLH

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_AllAttr);
end;
```


D10.2.132 TLBIALLHIS, TLB Invalidate All, Hyp mode, Inner Shareable

The TLBIALLHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIALLHIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLHIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_AllAttr);
end;

```

D10.2.133 TLBIALLIS, TLB Invalidate All, Inner Shareable

The TLBIALLIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk. The entries that are invalidated are as follows:

- If executed at EL1, all entries that:
 - Would be required for the EL1&0 translation regime.
 - Match the current VMID, if EL2 is implemented and enabled in the current Security state.
- If executed in Secure state when EL3 is using AArch32, all entries that would be required for the Secure PL1&0 translation regime.
- If executed at EL2, and if EL2 is enabled in the current Security state, the stage 1 or stage 2 translation table entries that would be required for the PL1&0 translation regime and matches the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIALLIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

`MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}`

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr);
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr);
end;

```

D10.2.134 TLBIALLNSNH, TLB Invalidate All, Non-Secure Non-Hyp

The TLBIALLNSNH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIALLNSNH is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLNSNH

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_AllAttr);
end;
```

D10.2.135 TLBIALLNSNHIS, TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable

The TLBIALLNSNHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for stage 1 or stage 2 of the Non-secure PL1&0 translation regime, regardless of the associated VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIALLNSNHIS is a 32-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Rt> is ignored.

Executing TLBIALLNSNHIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_AllAttr);
end;

```

D10.2.136 TLBIASID, TLB Invalidate by ASID match

The TLBIASID characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
 - Is from a level of lookup above the final level.
 - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIASID is a 32-bit System instruction.

Field descriptions



Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing TLBIASID

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr,
            ↪R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBI_AllAttr, R(t));
end;
```

D10.2.137 TLBIASIDIS, TLB Invalidate by ASID match, Inner Shareable

The TLBIASIDIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
 - Is from a level of lookup above the final level.
 - Is a non-global entry from the final level of lookup.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIASIDIS is a 32-bit System instruction.

Field descriptions



Bits [31:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries for non-global pages that match the ASID values will be affected by this System instruction.

Executing TLBIASIDIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr,
            ↪R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBI_AllAttr, R(t));
end;
```


D10.2.138 TLBIIPAS2, TLB Invalidate by Intermediate Physical Address, Stage 2

The TLBIIPAS2 characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

Configuration

Note

This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIIPAS2 is a 32-bit System instruction.

Field descriptions

31	28	27		0
RES0			IPA[39:12]	

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing TLBIIPAS2

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0100	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪ TLBI_AllAttr, R(t));
end;
```

D10.2.139 TLBIIPAS2IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable

The TLBIIPAS2IS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

Note

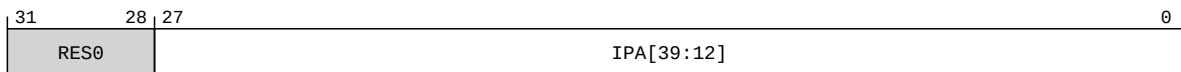
This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIIPAS2IS is a 32-bit System instruction.

Field descriptions



Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing TLBIIPAS2IS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```

D10.2.140 TLBIIPAS2L, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level

The TLBIIPAS2L characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

Configuration

Note

This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIIPAS2L is a 32-bit System instruction.

Field descriptions

31	28	27		0
RES0			IPA[39:12]	

Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing TLBIIPAS2L

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0100	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
end;
```

D10.2.141 TLBIIPAS2LIS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable

The TLBIIPAS2LIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- SCR.NS is 1.
- The entry would be used for the specified IPA.
- The entry would be used with the current VMID.
- The entry would be required for the PL1&0 translation regime.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

Note

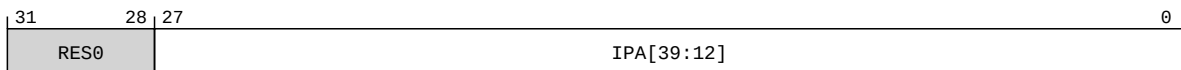
This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIIPAS2LIS is a 32-bit System instruction.

Field descriptions



Bits [31:28]

Reserved, RES0.

IPA[39:12], bits [27:0]

Bits[39:12] of the intermediate physical address to match.

Executing TLBIIPAS2LIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪ TLBI_AllAttr, R(t));
end;
```


D10.2.142 TLBIMVA, TLB Invalidate by VA

The TLBIMVA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
 - The entry is from a level of lookup above the final level and matches the specified ASID.
 - The entry is a global entry from the final level of lookup.
 - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This instruction is treated as a NOP.

Attributes

TLBIMVA is a 32-bit System instruction.

Field descriptions

31	12	11	8	7	0
VA			RES0	ASID	

VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Executing TLBIMVA

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```

D10.2.143 TLBIMVAA, TLB Invalidate by VA, All ASID

The TLBIMVAA characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

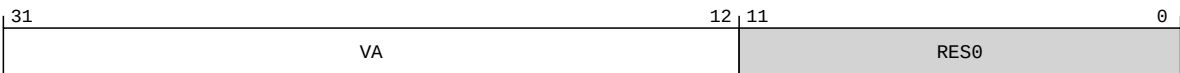
Configuration

This instruction is treated as a NOP.

Attributes

TLBIMVAA is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAA

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```

D10.2.144 TLBIMVAAIS, TLB Invalidate by VA, All ASID, Inner Shareable

The TLBIMVAAIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

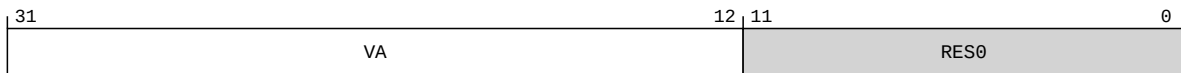
Configuration

This instruction is treated as a NOP.

Attributes

TLBIMVAAIS is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAAIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```

D10.2.145 TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level

The TLBIMVAAL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

Note

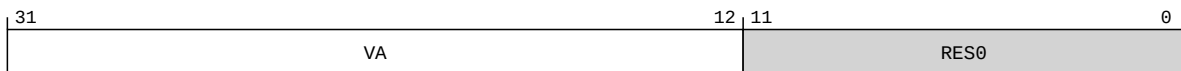
This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIMVAAL is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAAL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBIlevel_Last,
            ↪TLBI_AllAttr, R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBIlevel_Last,
        ↪TLBI_AllAttr, R(t));
end;
```


D10.2.146 TLBIMVAALIS, TLB Invalidate by VA, All ASID, Last level, Inner Shareable

The TLBIMVAALIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified address.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

Note

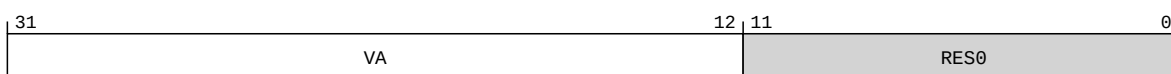
This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIMVAALIS is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any unlocked TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAALIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, R(t));
end;
```

D10.2.147 TLBIMVAH, TLB Invalidate by VA, Hyp mode

The TLBIMVAH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation only applies to the PE that executes this System instruction.

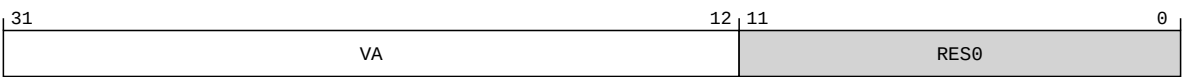
Configuration

This instruction is treated as a NOP.

Attributes

TLBIMVAH is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAH

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
```

```
        else
            Undefined();
        end;
    elsif PSTATE.EL == EL2 then
        AArch32_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBIlevel_Any,
            ↪ TLBI_AllAttr, R(t));
    end;
```

D10.2.148 TLBIMVAHIS, TLB Invalidate by VA, Hyp mode, Inner Shareable

The TLBIMVAHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from any level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

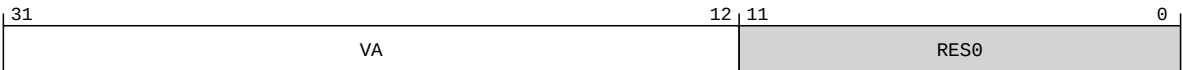
Configuration

This instruction is treated as a NOP.

Attributes

TLBIMVAHIS is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVAHIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBIlevel_Any,
        ↪ TLBI_AllAttr, R(t));
end;
```

D10.2.149 TLBIMVAIS, TLB Invalidate by VA, Inner Shareable

The TLBIMVAIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
 - The entry is from a level of lookup above the final level and matches the specified ASID.
 - The entry is a global entry from the final level of lookup.
 - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

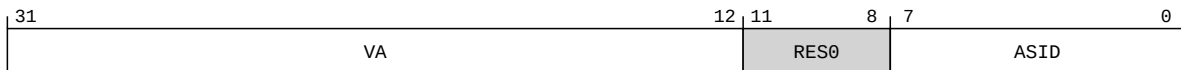
Configuration

This instruction is treated as a NOP.

Attributes

TLBIMVAIS is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Executing TLBIMVAIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, R(t));
end;
```


D10.2.150 TLBIMVAL, TLB Invalidate by VA, Last level

The TLBIMVAL characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
 - The entry is a global entry from the final level of lookup.
 - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIMVAL is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Executing TLBIMVAL

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBIlevel_Last,
            ↪ TLBI_AllAttr, R(t));
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_NSH, TLBIlevel_Last,
        ↪ TLBI_AllAttr, R(t));
end;
```

D10.2.151 TLBIMVALH, TLB Invalidate by VA, Last level, Hyp mode

The TLBIMVALH characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from the final level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation only applies to the PE that executes this System instruction.

Configuration

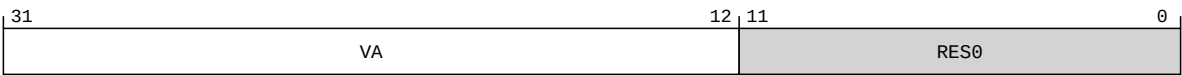
This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIMVALH is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVALH

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBIlevel_Last,
        ↪ TLBI_AllAttr, R(t));
end;
```

D10.2.152 TLBIMVALHIS, TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable

The TLBIMVALHIS characteristics are:

Purpose

If EL2 is implemented, invalidate all cached copies of translation table entries from TLBs that are from the final level of the translation table walk that would be required for the Non-secure EL2 translation regime and used to translate the specified address.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIMVALHIS is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:0]

Reserved, RES0.

Executing TLBIMVALHIS

If this instruction is executed in a Secure privileged mode other than Monitor mode, then the behavior is CONSTRAINED UNPREDICTABLE, and one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if it had been executed in Monitor mode.

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elseif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBIlevel_Last,
        ↪ TLBI_AllAttr, R(t));
end;
```

D10.2.153 TLBIMVALIS, TLB Invalidate by VA, Last level, Inner Shareable

The TLBIMVALIS characteristics are:

Purpose

Invalidate all cached copies of translation table entries from TLBs that meet the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified address, and one of the following applies:
 - The entry is a global entry from the final level of lookup.
 - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- If EL2 is implemented and enabled in the current Security state, the entry would be used with the current VMID.

From the entries that match these requirements, the entries that are invalidated are required for the following translation regime:

- If executed at Secure EL1 when EL3 is using AArch64, the Secure EL1&0 translation regime.
- If executed in Secure state when EL3 is using AArch32, the Secure PL1&0 translation regime.
- If executed in Non-secure state, the Non-secure PL1&0 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

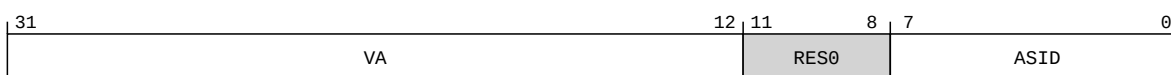
This System instruction is not implemented in architecture versions before Armv8.

This instruction is treated as a NOP.

Attributes

TLBIMVALIS is a 32-bit System instruction.

Field descriptions



VA, bits [31:12]

Virtual address to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Bits [11:8]

Reserved, RES0.

ASID, bits [7:0]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

Executing TLBIMVALIS

The following pseudocode describes traps which apply to the System instruction. For information about changes to the scope of the invalidation to the instruction under different conditions, see the relevant instruction in the [Pseudocode for AArch32 operation](#).

Accesses to this instruction use the following encodings in the System instruction encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T8 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
            ↪ TLBI_AllAttr, R(t));
    end;
elsif PSTATE.EL == EL2 then
    AArch32_TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID(), Broadcast_ISH, TLBILevel_Last,
        ↪ TLBI_AllAttr, R(t));
end;
```


D10.2.154 TLBTR, TLB Type Register

The TLBTR characteristics are:

Purpose

Provides information about the TLB implementation. The register must define whether the implementation provides separate instruction and data TLBs, or a unified TLB. Normally, the IMPLEMENTATION DEFINED information in this register includes the number of lockable entries in the TLB.

Configuration

There are no configuration notes.

Attributes

TLBTR is a 32-bit register.

Field descriptions

31		1	0
IMPLEMENTATION DEFINED			nU

IMPLEMENTATION DEFINED, bits [31:1]

IMPLEMENTATION DEFINED.

nU, bit [0]

Not Unified TLB. Indicates whether the implementation has a unified TLB.

The value of this field is an IMPLEMENTATION DEFINED choice of:

nU	Meaning
0b0	Unified TLB.
0b1	Separate Instruction and Data TLBs.

Access to this field is RO.

Accessing TLBTR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HCR().TID1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = TLBTR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = TLBTR();
end;
```

D10.2.155 TPIDRPRW, PL1 Software Thread ID Register

The TPIDRPRW characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is not visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configuration

Note

The PE never updates this register.

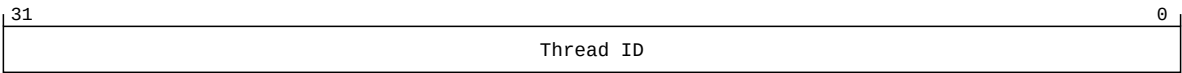
Attributes

TPIDRPRW is a 32-bit register.

This register has the following instances:

- TPIDRPRW.

Field descriptions



TID, bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDRPRW

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = TPIDRPRW();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = TPIDRPRW();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        TPIDRPRW() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    TPIDRPRW() = R(t);
end;
```

D10.2.156 TPIDRURO, PL0 Read-Only Software Thread ID Register

The TPIDRURO characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Configuration

Note

The PE never updates this register.

Attributes

TPIDRURO is a 32-bit register.

This register has the following instances:

- TPIDRURO.

Field descriptions

31	0
Thread ID	

TID, bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDRURO

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b011

```

if PSTATE.EL == EL0 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = TPIDRURO();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    end;
end;

```

```
        else
            R(t) = TPIDRURO();
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = TPIDRURO();
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b011

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        TPIDRURO() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    TPIDRURO() = R(t);
end;
```

D10.2.157 TPIDRURW, PL0 Read/Write Software Thread ID Register

The TPIDRURW characteristics are:

Purpose

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Configuration

Note

The PE never updates this register.

Attributes

TPIDRURW is a 32-bit register.

This register has the following instances:

- TPIDRURW.

Field descriptions

31	0
Thread ID	

TID, bits [31:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDRURW

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b010

```

if PSTATE.EL == EL0 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = TPIDRURW();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    end;
end;

```

```
        else
            R(t) = TPIDRURW();
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = TPIDRURW();
    end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        TPIDRURW() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T13 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        TPIDRURW() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    TPIDRURW() = R(t);
end;
```


D10.2.158 VBAR, Vector Base Address Register

The VBAR characteristics are:

Purpose

When high exception vectors are not selected, holds the vector base address for exceptions that are not taken to Monitor mode or to Hyp mode.

Software must program VBAR(NS) with the required initial value as part of the PE boot sequence.

Configuration

There are no configuration notes.

Attributes

VBAR is a 32-bit register.

This register has the following instances:

- VBAR.

Field descriptions

31	5	4	0
Vector Base Address			RES0

VBA, bits [31:5]

Vector Base Address. Bits[31:5] of the base address of the exception vectors for exceptions taken to this Exception level. Bits[4:0] of an exception vector are the exception offset.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bits [4:0]

Reserved, RES0.

Accessing VBAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T12 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = VBAR();
```

```
end;  
elseif PSTATE.EL == EL2 then  
    R(t) = VBAR();  
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then  
    Undefined();  
elseif PSTATE.EL == EL1 then  
    if EL2Enabled() && HSTR().T12 == '1' then  
        AArch32_TakeHypTrapException(0x03);  
    else  
        VBAR() = R(t);  
    end;  
elseif PSTATE.EL == EL2 then  
    VBAR() = R(t);  
end;
```

D10.2.159 VMPIDR, Virtualization Multiprocessor ID Register

The VMPIDR characteristics are:

Purpose

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by Non-secure EL1 reads of [MPIDR](#), which in a multiprocessor system, provides an additional PE identification system.

Configuration

There are no configuration notes.

Attributes

VMPIDR is a 32-bit register.

Field descriptions

31	30	29	25	24	23	16	15	8	7	0
M	U	RES0	MT	Aff2			Aff1		Aff0	

M, bit [31]

Indicates whether this implementation includes the functionality introduced by the Armv7 Multiprocessing Extensions.

M	Meaning
0b0	This implementation does not include the Armv7 Multiprocessing Extensions functionality.
0b1	This implementation includes the Armv7 Multiprocessing Extensions functionality.

Access to this field is RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MPIDR](#).U.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of Aff0 for more information about affinity levels.

MT	Meaning
0b0	Performance of PEs at the lowest affinity level is largely independent.
0b1	Performance of PEs at the lowest affinity level is very interdependent.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MPIDR.MT](#).

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MPIDR.Aff2](#).

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MPIDR.Aff1](#).

Aff0, bits [7:0]

Affinity level 0. The value of the [MPIDR](#).{Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MPIDR.Aff0](#).

Accessing VMPIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b101

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = VMPIDR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b101

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VMPIDR() = R(t);
end;

```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b101

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() then
        R(t) = VMPIDR();
    else
        R(t) = MPIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MPIDR();
end;

```

D10.2.160 VPIDR, Virtualization Processor ID Register

The VPIDR characteristics are:

Purpose

Holds the value of the Virtualization Processor ID. This is the value returned by Non-secure EL1 reads of [MIDR](#).

Configuration

There are no configuration notes.

Attributes

VPIDR is a 32-bit register.

Field descriptions

31	24	23	20	19	16	15	4	3	0
Implementer		Variant		Architecture		PartNum		Revision	

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MIDR](#).Implementer.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MIDR](#).Variant.

Architecture, bits [19:16]

Architecture version.

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MIDR](#).Architecture.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MIDR](#).PartNum.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [MIDR](#).Revision.

Accessing VPIDR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = VPIDR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0000	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VPIDR() = R(t);
end;

```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b0000	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T0 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() then
        R(t) = VPIDR();
    else
        R(t) = MIDR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = MIDR();
end;

```


D10.2.161 VSCTLR, Virtualization System Control Register

The VSCTLR characteristics are:

Purpose

Provides control and configuration information for PMSAv8-32 virtualization using stage 2 of EL1&0 translation regime.

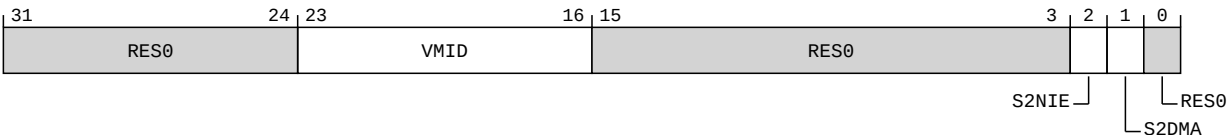
Configuration

There are no configuration notes.

Attributes

VSCTLR is a 32-bit register.

Field descriptions



Bits [31:24]

Reserved, RES0.

VMID, bits [23:16]

Virtual machine identifier.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:3]

Reserved, RES0.

S2NIE, bit [2]

Stage 2 Normal Interrupt Enable.

S2NIE	Meaning
0b0	This feature is disabled.
0b1	Multi-word accesses are interruptible, regardless of stage 1 attributes, where the access: - Uses base-restore addressing. - Is executed at EL1 or EL0. - Is an access to memory marked as Normal in the corresponding EL2 MPU region.

This field only has any effect when [HSCTLR.FI](#) is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

S2DMA, bit [1]

Stage 2 Device Multiword Access Disable.

S2DMA	Meaning
0b0	This feature is disabled.
0b1	Multi-word accesses at EL1 or EL0 that span an aligned 64-bit boundary generate a stage 2 Permission Fault if the memory region is marked as Device in the corresponding EL2 MPU region.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing VSCTLR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0000	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T2 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = VSCTLR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T2 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    VSCTLR() = R(t);
end;
```

D10.3 Debug System registers

This section lists the Debug System registers in AArch32 state, in alphabetic order.

D10.3.1 DBGAUTHSTATUS, Debug Authentication Status register

The DBGAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configuration

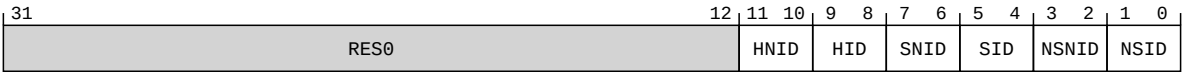
This register is required in all implementations.

AArch32 System register DBGAUTHSTATUS bits [31:0] are architecturally mapped to External register [DBGAUTHSTATUS_EL1](#) [31:0].

Attributes

DBGAUTHSTATUS is a 32-bit register.

Field descriptions



Bits [31:12]

Reserved, RES0.

HNID, bits [11:10]

Hyp non-invasive debug.

HNID	Meaning
0b00	Separate Hyp enable not implemented.
0b10	Implemented and disabled. ExternalHypNoninvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalHypNoninvasiveDebugEnabled() == TRUE.

Only permitted values for this field are 0b10 and 0b11. All other values are reserved.

HID, bits [9:8]

Hyp invasive debug.

HID	Meaning
0b00	Separate Hyp enable not implemented.
0b10	Implemented and disabled. ExternalHypInvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalHypInvasiveDebugEnabled() == TRUE.

Only permitted values for this field are 0b10 and 0b11. All other values are reserved.

SNID, bits [7:6]

Secure Non-Invasive Debug.

SNID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. <code>ExternalSecureNoninvasiveDebugEnabled()</code> == FALSE.
0b11	Implemented and enabled. <code>ExternalSecureNoninvasiveDebugEnabled()</code> == TRUE.

For Armv8-R AArch32, the only permitted value for this field is 0b00.

All other values are reserved.

SID, bits [5:4]

Secure Invasive Debug.

SID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. <code>ExternalSecureInvasiveDebugEnabled()</code> == FALSE.
0b11	Implemented and enabled. <code>ExternalSecureInvasiveDebugEnabled()</code> == TRUE.

For Armv8-R AArch32, the only permitted value for this field is 0b00.

All other values are reserved.

NSNID, bits [3:2]

Non-secure Non-Invasive Debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. <code>ExternalNoninvasiveDebugEnabled()</code> == FALSE.
0b11	Implemented and enabled. <code>ExternalNoninvasiveDebugEnabled()</code> == TRUE.

For Armv8-R AArch32, the only permitted values for this field are 0b10 and 0b11.

All other values are reserved.

NSID, bits [1:0]

Non-secure Invasive Debug.

NSID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

For Armv8-R AArch32, the only permitted values for this field are 0b10 and 0b11.
All other values are reserved.

Accessing DBGAUTHSTATUS

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1110	0b110

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGAUTHSTATUS();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGAUTHSTATUS();
end;
```

D10.3.2 DBGBCR<n>, Debug Breakpoint Control Registers, n = 0 - 15

The DBGBCR<n> characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register [DBGBVR<n>](#). If this breakpoint supports Context matching, [DBGBVR<n>](#) can be associated with a Breakpoint Extended Value Register [DBGBXVR<n>](#) for VMID matching.

Configuration

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

AArch32 System register DBGBCR<n> bits [31:0] are architecturally mapped to External register [DBGBCR<n>_EL1](#)[31:0].

Attributes

DBGBCR<n> is a 32-bit register.

Field descriptions

31	24	23	20	19	16	15	14	13	12	9	8	5	4	3	2	1	0
RES0		BT		LBN		SSC	HMC		RES0		BAS		RES0		PMC		E

When the E field is zero, all the other fields in the register are ignored.

Bits [31:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type. Possible values are:

BT	Meaning
0b0000	Unlinked instruction address match. DBGBVR<n> is the address of an instruction.
0b0001	As 0b0000, but linked to a Context matching breakpoint.
0b0010	Unlinked Context ID match. DBGBVR<n> .ContextID is compared against CONTEXTIDR .
0b0011	As 0b0010 with linking enabled.
0b0100	Unlinked instruction address mismatch. DBGBVR<n> is the address of an instruction to be stepped.
0b0101	As 0b0100, but linked to a Context matching breakpoint.
0b0110	Unlinked CONTEXTIDR match. DBGBVR<n> .ContextID is a Context ID compared against CONTEXTIDR .
0b0111	As 0b0110 with linking enabled.
0b1000	Unlinked VMID match. DBGBXVR<n> .VMID is a VMID compared against VSCTLR .VMID.
0b1001	As 0b1000 with linking enabled.
0b1010	Unlinked VMID and Context ID match. DBGBVR<n> .ContextID is a Context ID compared against CONTEXTIDR , and DBGBXVR<n> .VMID is a VMID compared against VSCTLR .VMID.

BT	Meaning
0b1011	As 0b1010 with linking enabled.
0b1100	Unlinked CONTEXTIDR_EL2 match. DBG BXVR<n> .ContextID2 is a Context ID compared against CONTEXTIDR_EL2.
0b1101	As 0b1100 with linking enabled.
0b1110	Unlinked Full Context ID match. DBG BVR<n> .ContextID is compared against CONTEXTIDR , and DBG BXVR<n> .ContextID2 is compared against CONTEXTIDR_EL2.
0b1111	As 0b1110 with linking enabled.

For more information on Breakpoints and their constraints, see ‘[Breakpoint exceptions](#)’ and ‘[Reserved DBGBCR<n>.BT values](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number. For Linked address matching breakpoints, this specifies the index of the context-matching breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

This field is ignored when the value of DBGBCR<n>.E is 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields.

For more information, see ‘[Execution conditions for which a breakpoint generates Breakpoint exceptions](#)’ and ‘[Reserved DBGBCR<n>.{SSC, HMC, PMC} values](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the SSC, bits [15:14] description.

For more information on the operation of the SSC, HMC, and PMC fields, see ‘[Execution conditions for which a breakpoint generates Breakpoint exceptions](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

Byte address select. Defines which halfwords an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0b0011	DBGBVR<n>	Use for T32 instructions
0b1100	DBGBVR<n>+2	Use for T32 instructions
0b1111	DBGBVR<n>	Use for A32 instructions

All other values are reserved. For more information, see [‘Reserved DBGBCR<n>.BAS values’](#).

For more information on using the BAS field in Address Match breakpoints, see [‘Using the BAS field in Address Match breakpoints’](#).

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

BAS	Step instruction at	Constraint for debuggers
0b0000	•	Use for a match anywhere breakpoint
0b0011	DBGBVR<n>	Use for T32 instructions
0b1100	DBGBVR<n>+2	Use for T32 instructions
0b1111	DBGBVR<n>	Use for A32 instructions

All other values are reserved. For more information, see [‘Reserved DBGBCR<n>.BAS values’](#).

For more information on using the BAS field in address mismatch breakpoints, see [‘Using the BAS field in Address Match breakpoints’](#).

For Context matching breakpoints, this field is RES1 and ignored.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see [‘Execution conditions for which a breakpoint generates Breakpoint exceptions’](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable breakpoint [DBGBVR<n>](#). Possible values are:

E	Meaning
0b0	Breakpoint disabled.
0b1	Breakpoint enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBCR<n>

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b101

```

let m:integer = UInt(CRm[3:0]);

if m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBCR(m);
    end;
elsif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBCR(m);
    end;
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b101

```
let m:integer = UInt(CRM[3:0]);

if m >= NUM_BREAKPOINTS then
    Undefined();
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elseif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR(m) = R(t);
    end;
elseif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR(m) = R(t);
    end;
end;
```

D10.3.3 DBGBVR<n>, Debug Breakpoint Value Registers, n = 0 - 15

The DBGBVR<n> characteristics are:

Purpose

Holds a value for use in breakpoint matching, either the virtual address of an instruction or a context ID. Forms breakpoint n together with control register [DBGBCR<n>](#). If this breakpoint supports Context matching, DBGBVR<n> can be associated with a Breakpoint Extended Value Register [DBGBXVR<n>](#) for VMID matching.

Configuration

How this register is interpreted depends on the value of [DBGBCR<n>](#).BT.

- When [DBGBCR<n>](#).BT is 0b0x0x, this register holds a virtual address.
- When [DBGBCR<n>](#).BT is 0bxx1x, this register holds a Context ID.

For other values of [DBGBCR<n>](#).BT, this register is RES0.

Some breakpoints might not support Context ID comparison. For more information, see the description of the [DBGDIDR.CTX_CMPs](#) field.

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

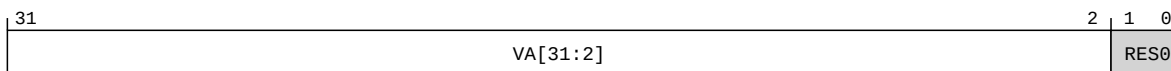
AArch32 System register DBGBVR<n> bits [31:0] are architecturally mapped to External register [DBGBVR<n>_EL1](#)[31:0].

Attributes

DBGBVR<n> is a 32-bit register.

Field descriptions

When [DBGBCR<n>](#).BT IN {'0x0x'}:



VA[31:2], bits [31:2]

Bits[31:2] of the address value for comparison.

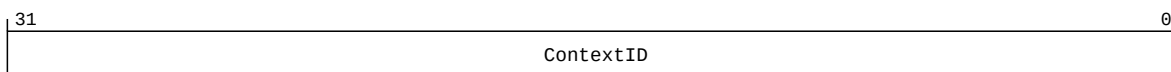
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When [DBGBCR<n>](#).BT IN {'001x'}:



ContextID, bits [31:0]

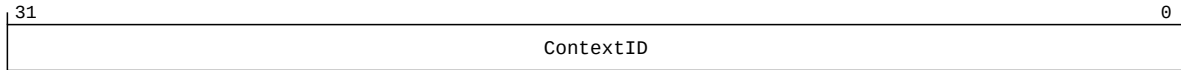
Context ID value for comparison.

The value is compared against [CONTEXTIDR](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>.BT IN {'101x'}:



ContextID, bits [31:0]

Context ID value for comparison against [CONTEXTIDR](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBVR<n>

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b100

```

let m:integer = UInt(CRm[3:0]);

if m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGBVR(m);
    end;
end;
end;

```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b100

```
let m:integer = UInt(CRm[3:0]);

if m >= NUM_BREAKPOINTS then
    Undefined();
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elseif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGGBVR(m) = R(t);
    end;
elseif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGGBVR(m) = R(t);
    end;
end;
```

D10.3.4 DBGBXVR<n>, Debug Breakpoint Extended Value Registers, n = 0 - 15

The DBGBXVR<n> characteristics are:

Purpose

Holds a value for use in breakpoint matching, to support VMID matching. Used in conjunction with a control register [DBGBCR<n>](#) and a value register [DBGBVR<n>](#), where breakpoint n supports Context matching.

Configuration

How this register is interpreted depends on the value of [DBGBCR<n>](#).BT.

- When [DBGBCR<n>](#).BT is 0b10xx, this register holds a VMID.
- When [DBGBCR<n>](#).BT is 0b11xx, this register holds a Context ID.

For other values of [DBGBCR<n>](#).BT, this register is RES0.

Accesses to this register are UNDEFINED in any of the following cases:

- Breakpoint n is not implemented.
- Breakpoint n does not support Context matching.

For more information, see the description of the [DBGDIDR](#).CTX_CMPs field.

AArch32 System register DBGBXVR<n> bits [31:0] are architecturally mapped to External register [DBGBVR<n>](#).EL1[63:32].

Attributes

DBGBXVR<n> is a 32-bit register.

Field descriptions

When [DBGBCR<n>](#).BT IN {'10xx'}:



Bits [31:8]

Reserved, RES0.

VMID[7:0], bits [7:0]

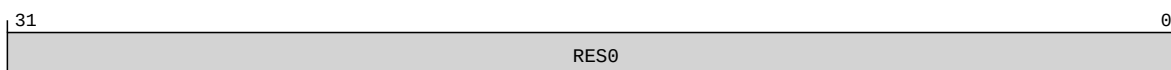
VMID value for comparison. The VMID is 8 bits when any of the following are true:

- EL2 is using AArch32.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When [DBGBCR<n>](#).BT IN {'11xx'}:



Bits [31:0]

Reserved, RES0.

Accessing DBGXVR<n>

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	m[3:0]	0b001

```
let m:integer = UInt(CRm[3:0]);

if m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGXVR(m);
    end;
elsif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGXVR(m);
    end;
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	m[3:0]	0b001

```
let m:integer = UInt(CRm[3:0]);

if m >= NUM_BREAKPOINTS then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGXVR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
```

```
if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
    Halt(DebugHalt_SoftwareAccess);
else
    DBG BXVR(m) = R(t);
end;
end;
```

D10.3.5 DBGCLAIMCLR, Debug CLAIM Tag Clear register

The DBGCLAIMCLR characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET register.

Configuration

An implementation must include eight CLAIM tag bits.

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMSET[31:0].

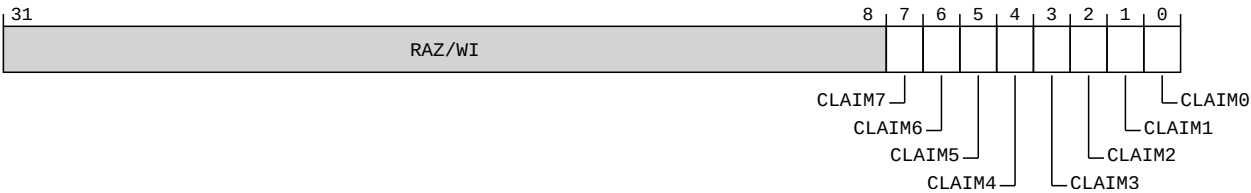
AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

AArch32 System register DBGCLAIMCLR bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

Attributes

DBGCLAIMCLR is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m>, bits [m], for m = 7 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is W1C.

Accessing DBGCLAIMCLR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1001	0b110

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGCLAIMCLR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGCLAIMCLR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1001	0b110

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGCLAIMCLR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    DBGCLAIMCLR() = R(t);
end;
```

D10.3.6 DBGCLAIMSET, Debug CLAIM Tag Set register

The DBGCLAIMSET characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.
The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR register.

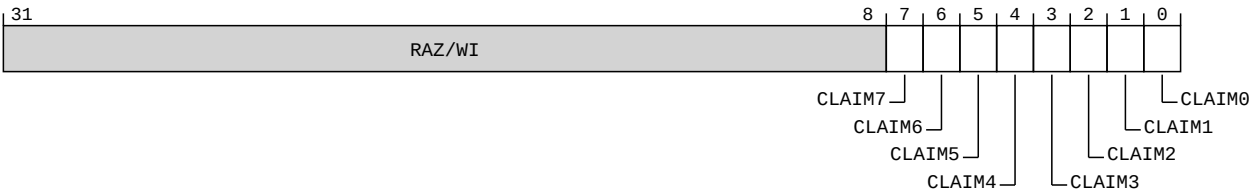
Configuration

An implementation must include eight CLAIM tag bits.
AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to AArch32 System register DBGCLAIMCLR[31:0].
AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].
AArch32 System register DBGCLAIMSET bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

Attributes

DBGCLAIMSET is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m>, bits [m], for m = 7 to 0

Claim Tag Set. Used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a write: Ignored.
0b1	On a write: Set Claim Tag bit <m> to 1.

Access to this field is RAO/WIS.

Accessing DBGCLAIMSET

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1000	0b110

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGCLAIMSET();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGCLAIMSET();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b1000	0b110

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGCLAIMSET() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    DBGCLAIMSET() = R(t);
end;

```

D10.3.7 DBGDCCINT, DCC Interrupt Enable Register

The DBGDCCINT characteristics are:

Purpose

Enables interrupt requests to be signaled based on the DCC status flags.

Configuration

There are no configuration notes.

Attributes

DBGDCCINT is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES0.

RX, bit [30]

DCC interrupt request enable control for DTRRX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

RX	Meaning
0b0	No interrupt request generated by DTRRX.
0b1	Interrupt request will be generated on RXfull == 1.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TX, bit [29]

DCC interrupt request enable control for DTRTX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

TX	Meaning
0b0	No interrupt request generated by DTRTX.

0b1	Interrupt request will be generated on TXfull == 0.
-----	---

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [28:0]

Reserved, RES0.

Accessing DBGDCCINT

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b000

```

if Halted() && PSTATE.EL != EL0 && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDCCINT();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDCCINT();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGDCCINT();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b000

```

if Halted() && PSTATE.EL != EL0 && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    DBGDCCINT() = R(t);
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGDCCINT() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    DBGDCCINT() = R(t);
end;

```


D10.3.8 **DBGDEVID, Debug Device ID register 0**

The DBGDEVID characteristics are:

Purpose

Adds to the information given by the [DBGDIDR](#) by describing other features of the debug implementation.

Configuration

This register is required in all implementations.

Attributes

DBGDEVID is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
CIDMask		AuxRegs		DoubleLock		VirtExtns		VectorCatch		BPAAddrMask		WPAAddrMask		PCSample	

CIDMask, bits [31:28]

Indicates the level of support for the Context ID matching breakpoint masking capability.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CIDMask	Meaning
0b0000	Context ID masking is not implemented.
0b0001	Context ID masking is implemented.

All other values are reserved. The value of this for Armv8 is 0b0000.

Access to this field is RO.

AuxRegs, bits [27:24]

Indicates support for Auxiliary registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AuxRegs	Meaning
0b0000	None supported.
0b0001	Support for External Debug Auxiliary Control Register, EDACR .

All other values are reserved.

Access to this field is RO.

DoubleLock, bits [23:20]

OS Double Lock implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DoubleLock	Meaning
0b0000	OS Double Lock is not implemented. DBGOSDLR is RAZ/WI.
0b0001	OS Double Lock is implemented. DBGOSDLR is RW.

[FEAT_DoubleLock](#) implements the functionality identified by the value 0b0001.

All other values are reserved.

Access to this field is RO.

VirtExtns, bits [19:16]

Indicates whether EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VirtExtns	Meaning
0b0000	EL2 is not implemented.
0b0001	EL2 is implemented.

All other values are reserved.

Access to this field is RO.

VectorCatch, bits [15:12]

Defines the form of Vector Catch exception implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VectorCatch	Meaning
0b0000	Address matching Vector Catch exception implemented.
0b0001	Exception matching Vector Catch exception implemented.

All other values are reserved.

Access to this field is RO.

BPAddrMask, bits [11:8]

Indicates the level of support for the instruction address matching breakpoint masking capability.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BPAAddrMask	Meaning
0b0000	Breakpoint address masking might be implemented. If not implemented, DBGBCR<n> [28:24] is RAZ/WI.
0b0001	Breakpoint address masking is implemented.
0b1111	Breakpoint address masking is not implemented. DBGBCR<n> [28:24] is RES0.

All other values are reserved. The value of this for Armv8 is 0b1111.

Access to this field is RO.

WPAAddrMask, bits [7:4]

Indicates the level of support for the data address matching watchpoint masking capability.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WPAAddrMask	Meaning
0b0000	Watchpoint address masking might be implemented. If not implemented, DBGWCR<n> .MASK (Address mask) is RAZ/WI.
0b0001	Watchpoint address masking is implemented.
0b1111	Watchpoint address masking is not implemented. DBGWCR<n> .MASK (Address mask) is RES0.

All other values are reserved. The value of this for Armv8 is 0b0001.

Access to this field is RO.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSample	Meaning
0b0000	PC Sample-based Profiling Extension is not implemented in the external debug registers space.
0b0010	Only EDPCSR and EDCIDS are implemented. This option is only permitted if EL3 and EL2 are not implemented.
0b0011	EDPCSR , EDCIDS , and EDVIDSR are implemented.

The table above shows all possible values of this field. The values this field can take in the current implementation are:

- 0b0010
- 0b0011

All other values are reserved.

Access to this field is RO.

Accessing DBGDEVID

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0010	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDEVID();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGDEVID();
end;
```

D10.3.9 DBGDEVID1, Debug Device ID register 1

The DBGDEVID1 characteristics are:

Purpose

Adds to the information given by the [DBGDIDR](#) by describing other features of the debug implementation.

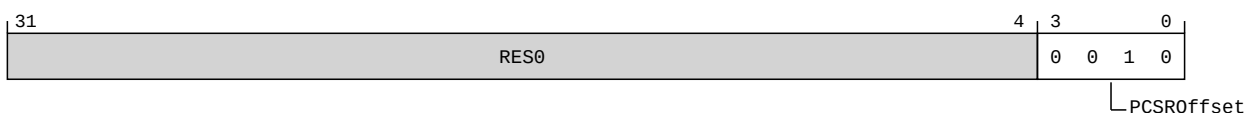
Configuration

This register is required in all implementations.

Attributes

DBGDEVID1 is a 32-bit register.

Field descriptions

**Bits [31:4]**

Reserved, RES0.

PCSROffset, bits [3:0]

This field indicates the offset applied to PC samples returned by reads of [EDPCR](#).

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSROffset	Meaning
0b0000	EDPCSR is not implemented.
0b0010	EDPCSR implemented. Samples have no offset applied and do not sample the instruction set state in AArch32 state.

The table above shows all possible values of this field. 0b0010 is the value of this field in the current implementation.

Access to this field is RO.

Accessing DBGDEVID1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0001	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDEVID1();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGDEVID1();
end;
```

D10.3.10 **DBGDEVID2, Debug Device ID register 2**

The DBGDEVID2 characteristics are:

Purpose

Reserved for future descriptions of features of the debug implementation.

Configuration

There are no configuration notes.

Attributes

DBGDEVID2 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing DBGDEVID2

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0111	0b0000	0b111

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDEVID2();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGDEVID2();
end;
```

D10.3.11 DBGDIDR, Debug ID Register

The DBGDIDR characteristics are:

Purpose

Specifies which version of the Debug architecture is implemented, and some features of the debug implementation.

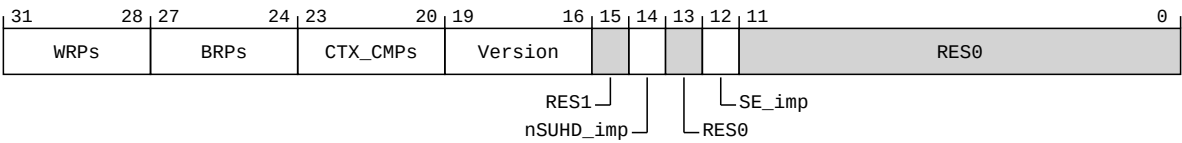
Configuration

If EL1 cannot use AArch32 then the implementation of this register is OPTIONAL and deprecated.

Attributes

DBGDIDR is a 32-bit register.

Field descriptions



WRPs, bits [31:28]

Number of watchpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WRPs	Meaning
0b0001..0b1111	The number of watchpoints, minus 1.

The value 0b0000 is reserved.

Access to this field is RO.

BRPs, bits [27:24]

Number of breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BRPs	Meaning
0b0001..0b1111	The number of breakpoints, minus 1.

The value 0b0000 is reserved.

Access to this field is RO.

CTX_CMPs, bits [23:20]

Number of context-aware breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CTX_CMPs	Meaning
0b0000..0b1111	The number of context-aware breakpoints, minus 1.

The value of this field is never greater than DBGDIDR.BRPs.

Access to this field is RO.

Version, bits [19:16]

Debug architecture version. Indicates presence of Armv8 debug architecture.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

From Armv8.0, the values 0b0000, 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101 are not permitted.

Access to this field is RO.

Bit [15]

Reserved, RES1.

nSUHD_imp, bit [14]

Previously indicated that Secure User Halting Debug is not implemented.

The value of this field must match the value of the SE_imp field.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [13]

Reserved, RES0.

SE_imp, bit [12]

EL3 implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SE_imp	Meaning
0b0	EL3 not implemented.
0b1	EL3 implemented.

The value of this field must match the value of the nSUHD_imp field.

Access to this field is RO.

Bits [11:0]

Reserved, RES0.

Accessing DBGDIDR

Arm deprecates any access to this register from EL0.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b000

```
if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDIDR();
elseif PSTATE.EL == EL0 then
    if DBGDSCRExt().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDIDR();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDIDR();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = DBGDIDR();
end;
```

D10.3.12 DBGDRAR, Debug ROM Address Register

The DBGDRAR characteristics are:

Purpose

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. Use of this register is deprecated.

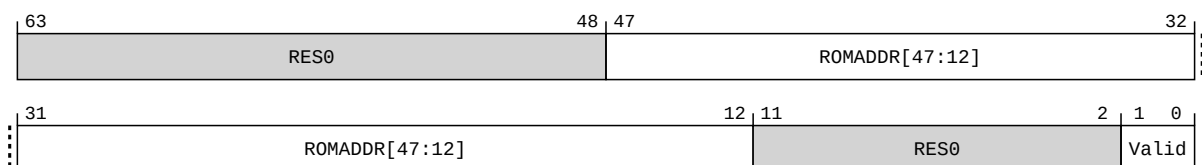
Configuration

DBGDRAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

Attributes

DBGDRAR is a 64-bit register.

Field descriptions



Bits [63:48]

Reserved, RES0.

ROMADDR[47:12], bits [47:12]

Bits[47:12] of the ROM table physical address.

If the physical address size in bits (PAsize) is less than 48 then the register bits corresponding to ROMADDR[47:PAsize] are RES0.

Bits [11:0] of the ROM table physical address are zero.

Arm strongly recommends that bits ROMADDR[(PAsize-1):32] are zero in any system where the implementation only supports execution in AArch32 state.

If DBGDRAR.Valid == 0b00, then this field is UNKNOWN.

In an implementation that includes EL3, ROMADDR is an address in Non-secure memory. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure memory.

Bits [11:2]

Reserved, RES0.

Valid, bits [1:0]

This field indicates whether the ROM Table address is valid.

Valid	Meaning
0b00	ROM Table address is not valid. Software must ignore ROMADDR.
0b11	ROM Table address is valid.

Other values are reserved.

Arm recommends implementations set this field to zero.

Accessing DBGDRAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0000	0b000

```
if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDRAR() [31:0];
elsif PSTATE.EL == EL0 then
    if DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDRA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDRAR() [31:0];
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDRA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDRAR() [31:0];
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGDRAR() [31:0];
end;
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1110	0b0001	0b0000

```
if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t, t2) = DBGDRAR();
elsif PSTATE.EL == EL0 then
    if DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
```

```
        AArch32_TakeHypTrapException(0x00);
    else
        Undefined();
    end;
end;
elsif EL2Enabled() && (HCR().TGE == '1' || HCR().[TDE,TDRA] != '00') then
    AArch32_TakeHypTrapException(0x0C);
else
    R(t, t2) = DBGDRAR();
end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR().[TDE,TDRA] != '00' then
        AArch32_TakeHypTrapException(0x0C);
    else
        R(t, t2) = DBGDRAR();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = DBGDRAR();
end;
```

D10.3.13 DBGDSAR, Debug Self Address Register

The DBGDSAR characteristics are:

Purpose

In earlier versions of the Arm Architecture, this register defines the offset from the base address defined in [DBGDRAR](#) of the physical base address of the debug registers for the PE. Use of this register is deprecated.

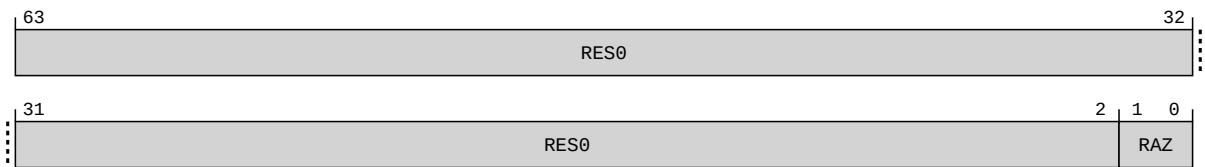
Configuration

DBGDSAR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, bits [31:0] are read.

Attributes

DBGDSAR is a 64-bit register.

Field descriptions



Bits [63:2]

Reserved, RES0.

Bits [1:0]

Reserved, RAZ.

This field indicates whether the debug self address offset is valid. For ARMv8, this field is always 0b00, the offset is not valid.

Accessing DBGDSAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0010	0b0000	0b000

```
if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDSAR() [31:0];
elsif PSTATE.EL == EL0 then
    if DBGDSCRExt().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    end;
```

```
        elsif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDRA] != '00') then
            AArch32_TakeHypTrapException(0x05);
        else
            R(t) = DBGDSAR()[31:0];
        end;
    elsif PSTATE.EL == EL1 then
        if EL2Enabled() && HDCR().[TDE,TDRA] != '00' then
            AArch32_TakeHypTrapException(0x05);
        else
            R(t) = DBGDSAR()[31:0];
        end;
    elsif PSTATE.EL == EL2 then
        R(t) = DBGDSAR()[31:0];
    end;
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1110	0b0010	0b0000

```
if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t, t2) = DBGDSAR();
elsif PSTATE.EL == EL0 then
    if DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDRA] != '00') then
        AArch32_TakeHypTrapException(0x0C);
    else
        R(t, t2) = DBGDSAR();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDRA] != '00' then
        AArch32_TakeHypTrapException(0x0C);
    else
        R(t, t2) = DBGDSAR();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = DBGDSAR();
end;
```

D10.3.14 DBGDSCRext, Debug Status and Control Register, External View

The DBGDSCRext characteristics are:

Purpose

Main control register for the debug implementation.

Configuration

This register is required in all implementations.

AArch32 System register DBGDSCRext bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to External register [EDSCR](#)[31:29, 27:26, 23:21, 19, 14, 6].

AArch32 System register DBGDSCRext bit [15] is architecturally mapped to AArch32 System register [DBGDSCRint](#)[15].

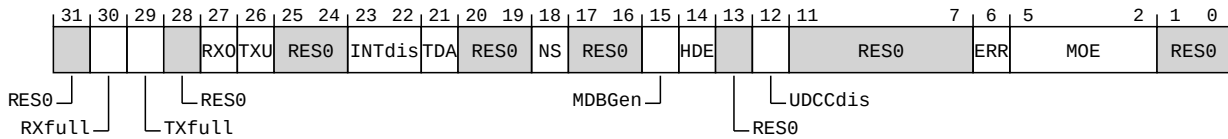
AArch32 System register DBGDSCRext bit [12] is architecturally mapped to AArch32 System register [DBGDSCRint](#)[12].

AArch32 System register DBGDSCRext bits [5:2] are architecturally mapped to AArch32 System register [DBGDSCRint](#)[5:2].

Attributes

DBGDSCRext is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full. Used for save/restore of [EDSCR](#).RXfull.

When [DBGOSLSR](#).OSLK = 0, software must treat this bit as UNK/SBZP.

When [DBGOSLSR](#).OSLK = 1, this bit holds the value of [EDSCR](#).RXfull. Reads and writes of this bit are indirect accesses to [EDSCR](#).RXfull.

Arm deprecates use of this bit other than for save/restore. Use [DBGDSCRint](#) to access the DTRRX full status.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When [DBGOSLSR](#).OSLK = '1', access to this field is RW.
- When [DBGOSLSR](#).OSLK = '0', access to this field is RO.

TXfull, bit [29]

DTRTX full. Used for save/restore of [EDSCR](#).TXfull.

When [DBGOSLSR](#).OSLK = 0, software must treat this bit as UNK/SBZP.

When `DBGOSLSR.OSLK == 1`, this bit holds the value of `EDSCR.TXfull`. Reads and writes of this bit are indirect accesses to `EDSCR.TXfull`.

Arm deprecates use of this bit other than for save/restore. Use `DBGDSCRint` to access the DTRTX full status.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When `DBGOSLSR.OSLK == '1'`, access to this field is RW.
- When `DBGOSLSR.OSLK == '0'`, access to this field is RO.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of `EDSCR.RXO`.

When `DBGOSLSR.OSLK == 0`, software must treat this bit as UNK/SBZP.

When `DBGOSLSR.OSLK == 1`, this bit holds the value of `EDSCR.RXO`. Reads and writes of this bit are indirect accesses to `EDSCR.RXO`.

When `DBGOSLSR.OSLK == 1`, if bits [27,6] of the value written to `DBGDSCRExt` are {1,0}, that is, the RXO bit is 1 and the ERR bit is 0, the PE sets `EDSCR.{RXO,ERR}` to UNKNOWN values.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When `DBGOSLSR.OSLK == '1'`, access to this field is RW.
- When `DBGOSLSR.OSLK == '0'`, access to this field is RO.

TXU, bit [26]

Used for save/restore of `EDSCR.TXU`.

When `DBGOSLSR.OSLK == 0`, software must treat this bit as UNK/SBZP.

When `DBGOSLSR.OSLK == 1`, this bit holds the value of `EDSCR.TXU`. Reads and writes of this bit are indirect accesses to `EDSCR.TXU`.

When `DBGOSLSR.OSLK == 1`, if bits [26,6] of the value written to `DBGDSCRExt` are {1,0}, that is, the TXU bit is 1 and the ERR bit is 0, the PE sets `EDSCR.{TXU,ERR}` to UNKNOWN values.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When `DBGOSLSR.OSLK == '1'`, access to this field is RW.
- When `DBGOSLSR.OSLK == '0'`, access to this field is RO.

Bits [25:24]

Reserved, RES0.

INTdis, bits [23:22]

Used for save/restore of `EDSCR.INTdis`.

When `DBGOSLSR.OSLK == 0`, software must treat it as UNK/SBZP.

When `DBGOSLSR.OSLK == 1`, this field holds the value of `EDSCR.INTdis`. Reads and writes of this field are indirect accesses to `EDSCR.INTdis`.

The reset behavior of this field is:

- On a Cold reset, this field resets to '00'.

Accessing this field has the following behavior:

- When `DBGOSLSR.OSLK == '1'`, access to this field is RW.
- When `DBGOSLSR.OSLK == '0'`, access to this field is RO.

TDA, bit [21]

Used for save/restore of `EDSCR.TDA`.

When `DBGOSLSR.OSLK == 0`, software must treat this bit as UNK/SBZP.

When `DBGOSLSR.OSLK == 1`, this bit holds the value of `EDSCR.TDA`. Reads and writes of this bit are indirect accesses to `EDSCR.TDA`.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When `DBGOSLSR.OSLK == '1'`, access to this field is RW.
- When `DBGOSLSR.OSLK == '0'`, access to this field is RO.

Bits [20:19]

Reserved, RES0.

NS, bit [18]

Non-secure status.

Arm deprecates use of this field.

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

Access to this field is RAO/WI.

Bits [17:16]

Reserved, RES0.

MDBGen, bit [15]

Monitor debug events enable. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

MDBGen	Meaning
0b0	Breakpoint, Watchpoint, and Vector Catch exceptions disabled.

0b1	Breakpoint, Watchpoint, and Vector Catch exceptions enabled.
-----	--

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

HDE, bit [14]

Used for save/restore of [EDSCR.HDE](#).

When [DBGOSLSR.OSLK](#) == 0, software must treat this bit as UNK/SBZP.

When [DBGOSLSR.OSLK](#) == 1, this bit holds the value of [EDSCR.HDE](#). Reads and writes of this bit are indirect accesses to [EDSCR.HDE](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When [DBGOSLSR.OSLK](#) == '1', access to this field is RW.
- When [DBGOSLSR.OSLK](#) == '0', access to this field is RO.

Bit [13]

Reserved, RES0.

UDCCdis, bit [12]

Traps EL0 accesses to the DCC registers to Undefined mode.

UDCCdis	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL0 accesses to the DBGDSCRint , DBGDTRRXint , DBGDTRTXint , DBGDIDR , DBGDSAR , and DBGDRAR are trapped to Undefined mode.

Note

All accesses to these registers are trapped, including LDC and STC accesses to [DBGDTRTXint](#) and [DBGDTRRXint](#), and MRRC accesses to [DBGDSAR](#) and [DBGDRAR](#).

Traps of EL0 accesses to the [DBGDTRRXint](#) and [DBGDTRTXint](#) are ignored in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of [EDSCR.ERR](#).

When [DBGOSLSR.OSLK](#) == 0, software must treat this bit as UNK/SBZP.

When [DBGOSLSR.OSLK](#) == 1, this bit holds the value of [EDSCR.ERR](#). Reads and writes of this bit are indirect accesses to [EDSCR.ERR](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When DBGOSLSR.OSLK == '1', access to this field is RW.
- When DBGOSLSR.OSLK == '0', access to this field is RO.

MOE, bits [5:2]

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

MOE	Meaning
0b0001	Breakpoint.
0b0011	Software breakpoint (BKPT) instruction.
0b0101	Vector catch.
0b1010	Watchpoint.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGDSCRext

Individual fields within this register might have restricted accessibility when the OS Lock is unlocked, [DBGOSLSR.OSLK](#) == 0. See the field descriptions for more detail.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b010

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDSCRext();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGDSCRext();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGDSCRext() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    DBGDSCRext() = R(t);
end;
```

D10.3.15 DBGDSCRint, Debug Status and Control Register, Internal View

The DBGDSCRint characteristics are:

Purpose

Main control register for the debug implementation. This is an internal, read-only view.

Configuration

This register is required in all implementations.

DBGDSCRint.{NS, SPNIDdis, SPIDdis, MDBGen, UDCCdis, MOE} are UNKNOWN when the register is accessed at EL0. However, although these values are not accessible at EL0 by instructions that are neither UNPREDICTABLE nor return UNKNOWN values, it is permissible for an implementation to return the values of DBGDSCRext.{NS, SPNIDdis, SPIDdis, MDBGen, UDCCdis, MOE} for these fields at EL0.

It is also permissible for an implementation to return the same values as defined for a read of `DBGDSCRint` at EL1 or above.

AArch32 System register DBGDSCRint bits [30:29] are architecturally mapped to External register [EDSCR](#)[30:29].

AArch32 System register DBGDSCRint bit [15] is architecturally mapped to AArch32 System register [DBGDSCRext](#)[15].

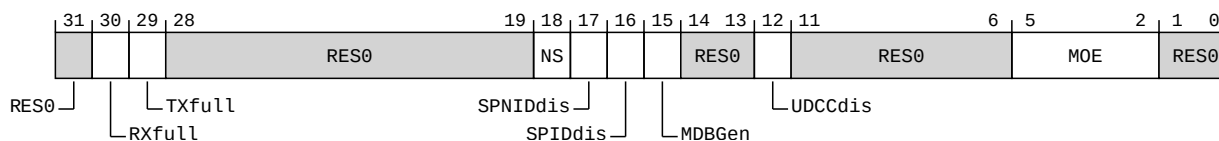
AArch32 System register `DBGDSCRint` bit [12] is architecturally mapped to AArch32 System register `DBGDSCRext`[12].

AArch32 System register DBGDSCRint bits [5:2] are architecturally mapped to AArch32 System register [DBGDSCRext](#)[5:2].

Attributes

DBGDSCRint is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full. Read-only view of the equivalent bit in the [EDSCR](#).

TXfull, bit [29]

DTRTX full. Read-only view of the equivalent bit in the [EDSCR](#).**Bits [28:19]**

Reserved, RES0.

NS, bit [18]

Non-secure status.

Read-only view of the equivalent bit in the [DBGDSCRext](#). Arm deprecates use of this field.

SPNIDdis, bit [17]

Secure privileged non-invasive debug disable.

Read-only view of the equivalent bit in the [DBGDSCRExt](#). Arm deprecates use of this field.

SPIDdis, bit [16]

Secure privileged invasive debug disable.

Read-only view of the equivalent bit in the [DBGDSCRExt](#). Arm deprecates use of this field.

MDBGen, bit [15]

Monitor debug events enable.

Read-only view of the equivalent bit in the [DBGDSCRExt](#).

Bits [14:13]

Reserved, RES0.

UDCCdis, bit [12]

User mode access to Debug Communications Channel disable.

Read-only view of the equivalent bit in the [DBGDSCRExt](#). Arm deprecates use of this field.

Bits [11:6]

Reserved, RES0.

MOE, bits [5:2]

Method of Entry for debug exception. When a debug exception is taken to an Exception level using AArch32, this field is set to indicate the event that caused the exception:

MOE	Meaning
0b0001	Breakpoint
0b0011	Software breakpoint (BKPT) instruction
0b0101	Vector catch
0b1010	Watchpoint

Read-only view of the equivalent bit in the [DBGDSCRExt](#).

Bits [1:0]

Reserved, RES0.

Accessing DBGDSCRint

When <Rt> is APSR_nzcv, encoded as R15, then instead of reading the entire register, the access copies DBGDSCRint[31:28] into the PSTATE NZCV flags.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0001	0b000

```

if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    if t == 15 then
        ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
    else
        R(t) = DBGDSCRint();
    end;
elsif PSTATE.EL == EL0 then
    if DBGDSCRExt().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    else
        if t == 15 then
            if Halted() then
                ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
            else
                PSTATE.[N,Z,C,V] = DBGDSCRint()[31:28];
            end;
        else
            R(t) = DBGDSCRint();
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        if t == 15 then
            if Halted() then
                ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
            else
                PSTATE.[N,Z,C,V] = DBGDSCRint()[31:28];
            end;
        else
            R(t) = DBGDSCRint();
        end;
    end;
elsif PSTATE.EL == EL2 then
    if t == 15 then
        if Halted() then
            ConstrainUnpredictableProcedure(Unpredictable_MRC_APSR_TARGET);
        else
            PSTATE.[N,Z,C,V] = DBGDSCRint()[31:28];
        end;
    else
        R(t) = DBGDSCRint();
    end;
end;
end;

```


D10.3.16 DBGDTRRXext, Debug OS Lock Data Transfer Register, Receive, External View

The DBGDTRRXext characteristics are:

Purpose

Used for save/restore of [DBGDTRRXint](#). It is a component of the Debug Communications Channel.

Configuration

There are no configuration notes.

Attributes

DBGDTRRXext is a 32-bit register.

Field descriptions

31	0
Update DTRRX without side-effect	

DTRRX, bits [31:0]

Update DTRRX without side-effect.

Writes to this register update the value in DTRRX and do not change RXfull.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see ‘[The Debug Communication Channel and Instruction Transfer Register](#)’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRXext

Arm deprecates reads and writes of DBGDTRRXext through the System register interface when the OS Lock is unlocked, [DBGOSLSR](#).OSLK == 0.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b010

```

if Halted() && PSTATE.EL != EL0 && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDTRRXext();
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDTRRXext();

```

```
end;  
elseif PSTATE.EL == EL2 then  
    R(t) = DBGDTRRXext();  
end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0000	0b010

```
if Halted() && PSTATE.EL != EL0 && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then  
    DBGDTRRXext() = R(t);  
elseif PSTATE.EL == EL0 then  
    Undefined();  
elseif PSTATE.EL == EL1 then  
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then  
        AArch32_TakeHypTrapException(0x05);  
    else  
        DBGDTRRXext() = R(t);  
    end;  
elseif PSTATE.EL == EL2 then  
    DBGDTRRXext() = R(t);  
end;
```

D10.3.17 DBGDTRRXint, Debug Data Transfer Register, Receive

The DBGDTRRXint characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

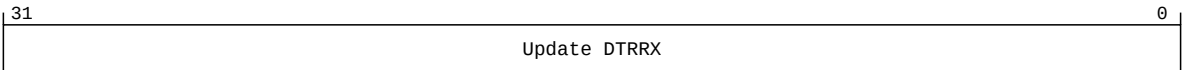
Configuration

AArch32 System register DBGDTRRXint bits [31:0] are architecturally mapped to External register [DBGDTRRX_EL0](#)[31:0].

Attributes

DBGDTRRXint is a 32-bit register.

Field descriptions



DTRRX, bits [31:0]

Update DTRRX.

Reads of this register:

- If RXfull is 1, return the last value written to DTRRX.
- If RXfull is 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

For the full behavior of the Debug Communications Channel, see ‘[The Debug Communication Channel and Instruction Transfer Register](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRXint

Data can be stored to memory from this register using [STC](#).

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0101	0b000

```
if Halted() then
    R(t) = Read_DBGDTR_EL0{32}();
elseif PSTATE.EL == EL0 then
    if DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = Read_DBGDTR_EL0{32}();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = Read_DBGDTR_EL0{32}();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = Read_DBGDTR_EL0{32}();
end;
```

STC{<c>}{<q>} <coproc>, <CRd>, <addressing_mode>

coproc	CRd
0b1110	0b0101

```
if Halted() then
    MemA{32}(address) = Read_DBGDTR_EL0{32}();
elseif PSTATE.EL == EL0 then
    if DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x06);
    else
        MemA{32}(address) = Read_DBGDTR_EL0{32}();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x06);
    else
        MemA{32}(address) = Read_DBGDTR_EL0{32}();
    end;
elseif PSTATE.EL == EL2 then
    MemA{32}(address) = Read_DBGDTR_EL0{32}();
end;
```

D10.3.18 DBGDTRTXext, Debug OS Lock Data Transfer Register, Transmit

The DBGDTRTXext characteristics are:

Purpose

Used for save/restore of [DBGDTRTXint](#). It is a component of the Debug Communication Channel.

Configuration

There are no configuration notes.

Attributes

DBGDTRTXext is a 32-bit register.

Field descriptions

31	0
Return DTRTX without side-effect	

DTRTX, bits [31:0]

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see ‘[The Debug Communication Channel and Instruction Transfer Register](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTXext

Arm deprecates reads and writes of DBGDTRTXext through the System register interface when the OS Lock is unlocked, [DBGOSLSR](#).OSLK == 0.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0011	0b010

```

if Halted() && PSTATE.EL != EL0 && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    R(t) = DBGDTRTXext();
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGDTRTXext();

```

```
        end;  
    elsif PSTATE.EL == EL2 then  
        R(t) = DBGDTRTXext();  
    end;
```

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0011	0b010

```
if Halted() && PSTATE.EL != EL0 && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then  
    DBGDTRTXext() = R(t);  
elseif PSTATE.EL == EL0 then  
    Undefined();  
elseif PSTATE.EL == EL1 then  
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then  
        AArch32_TakeHypTrapException(0x05);  
    else  
        DBGDTRTXext() = R(t);  
    end;  
elseif PSTATE.EL == EL2 then  
    DBGDTRTXext() = R(t);  
end;
```

D10.3.19 DBGDTRTXint, Debug Data Transfer Register, Transmit

The DBGDTRTXint characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

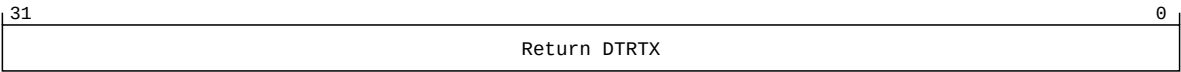
Configuration

AArch32 System register DBGDTRTXint bits [31:0] are architecturally mapped to External register [DBGDTRTX_EL0](#)[31:0].

Attributes

DBGDTRTXint is a 32-bit register.

Field descriptions



DTRTX, bits [31:0]

DTRTX. Writes to this register:

- If TXfull is 1, DTRTX is set to an UNKNOWN value.
- If TXfull is 0, update the value in DTRTX.

After the write, TXfull is set to 1.

For the full behavior of the Debug Communications Channel, see ‘[The Debug Communication Channel and Instruction Transfer Register](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTXint

Data can be loaded from memory into this register using ‘LDC (immediate)’ and ‘LDC (literal)’.

Accesses to this register use the following encodings in the System register encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0101	0b000

```
if Halted() then
    Write_DBGDTR_EL0{32}(R(t));
elseif PSTATE.EL == EL0 then
    if DBGDSCRExt().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
```

```
        else
            Undefined();
        end;
    elsif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x05);
    else
        Write_DBGDTR_EL0{32}(R(t));
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        Write_DBGDTR_EL0{32}(R(t));
    end;
elseif PSTATE.EL == EL2 then
    Write_DBGDTR_EL0{32}(R(t));
end;
```

LDC{<c>}{<q>} <coproc>, <CRd>, <addressing_mode>

coproc	CRd
0b1110	0b0101

```
if Halted() then
    Write_DBGDTR_EL0{32}(MemA{32}(address));
elseif PSTATE.EL == EL0 then
    if DBGDSCRext().UDCCdis == '1' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && (HCR().TGE == '1' || HDCR().[TDE,TDA] != '00') then
        AArch32_TakeHypTrapException(0x06);
    else
        Write_DBGDTR_EL0{32}(MemA{32}(address));
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x06);
    else
        Write_DBGDTR_EL0{32}(MemA{32}(address));
    end;
elseif PSTATE.EL == EL2 then
    Write_DBGDTR_EL0{32}(MemA{32}(address));
end;
```


D10.3.20 **DBGOSDLR, Debug OS Double Lock Register**

The DBGOSDLR characteristics are:

Purpose

Locks out the external debug interface.

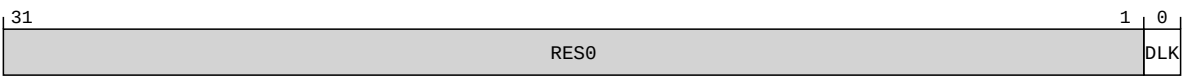
Configuration

There are no configuration notes.

Attributes

DBGOSDLR is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

DLK, bit [0]

When FEAT_DoubleLock is implemented:

OS Double Lock control bit.

DLK	Meaning
0b0	OS Double Lock unlocked.
0b1	OS Double Lock locked, if DBGPRCR.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RAZ/WI.

Accessing DBGOSDLR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0011	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDOSA] != '00' && (IsFeatureImplemented(FEAT_DoubleLock) ||
        ↪ImpDefBool("Trapped by HDCR.TDOSA")) then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGOSDLR();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = DBGOSDLR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0011	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDOSA] != '00' && (IsFeatureImplemented(FEAT_DoubleLock) ||
        ↪ImpDefBool("Trapped by HDCR.TDOSA")) then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGOSDLR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    DBGOSDLR() = R(t);
end;

```

D10.3.21 DBGOSECCR, Debug OS Lock Exception Catch Control Register

The DBGOSECCR characteristics are:

Purpose

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

Configuration

If DBGOSLSR.OSLK == 0 then DBGOSECCR returns an UNKNOWN value on reads and ignores writes.

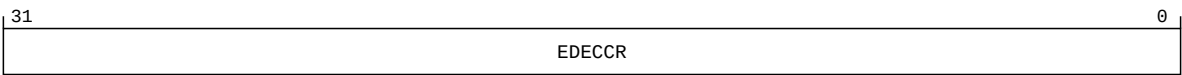
AArch32 System register DBGOSECCR bits [31:0] are architecturally mapped to External register EDECCR[31:0].

Attributes

DBGOSECCR is a 32-bit register.

Field descriptions

When DBGOSLSR.OSLK == '1':



EDECCR, bits [31:0]

Used for save/restore to EDECCR over powerdown.

Reads or writes to this field are indirect accesses to EDECCR.

The reset behavior of this field is:

- On a Cold reset, this field resets to the expression 0x00.

Accessing DBGOSECCR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b010

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif DBGOSLSR().OSLK == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = DBGOSECCR();
    end;
elsif PSTATE.EL == EL2 then
```

```
        if DBGOSLSR().OSLK == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = DBGOSECCR();
        end;
    end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elsif DBGOSLSR().OSLK == '0' then
        return;
    else
        DBGOSECCR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' then
        return;
    else
        DBGOSECCR() = R(t);
    end;
end;
```

D10.3.22 **DBGOSLAR, Debug OS Lock Access Register**

The DBGOSLAR characteristics are:

Purpose

Provides a lock for the debug registers. The OS Lock also disables some debug exceptions and debug events.

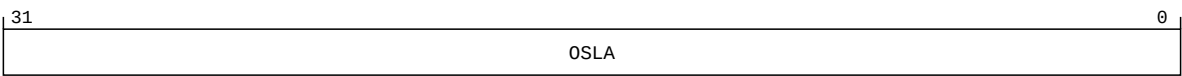
Configuration

There are no configuration notes.

Attributes

DBGOSLAR is a 32-bit register.

Field descriptions



OSLA, bits [31:0]

OS Lock Access. Writing the value 0xC5ACCE55 to the DBGOSLAR sets the OS Lock to 1. Writing any other value sets the OS Lock to 0.

Use [DBGOSLSR.OSLK](#) to check the current status of the lock.

Accessing DBGOSLAR

Accesses to this register use the following encodings in the System register encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0000	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDOSA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGOSLAR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    DBGOSLAR() = R(t);
end;
```

D10.3.23 DBGOSLSR, Debug OS Lock Status Register

The DBGOSLSR characteristics are:

Purpose

Provides status information for the OS Lock.

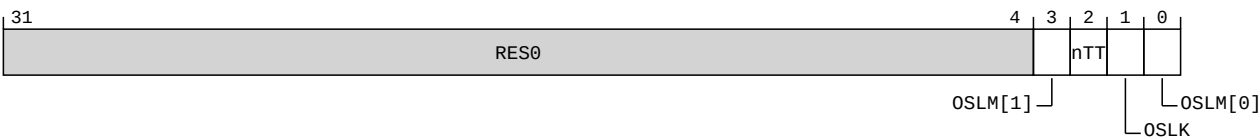
Configuration

The OS Lock status is also visible in the external debug interface through EDPRSR.

Attributes

DBGOSLSR is a 32-bit register.

Field descriptions



Bits [31:4]

Reserved, RES0.

OSLM, bits [3, 0]

OS Lock model implemented. Identifies the form of OS save and restore mechanism implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OSLM	Meaning
0b00	OS Lock not implemented.
0b10	OS Lock implemented.

All other values are reserved. In an Armv8 implementation the value 0b00 is not permitted.

Access to this field is RO.

nTT, bit [2]

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

OSLK, bit [1]

OS Lock Status.

OSLK	Meaning
0b0	OS Lock unlocked.
0b1	OS Lock locked.

The OS Lock is locked and unlocked by writing to the OS Lock Access Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to '1'.

Accessing DBGOSLSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDOSA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGOSLSR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGOSLSR();
end;
```

D10.3.24 DBGPRCR, Debug Power Control Register

The DBGPRCR characteristics are:

Purpose

Controls behavior of the PE on powerdown request.

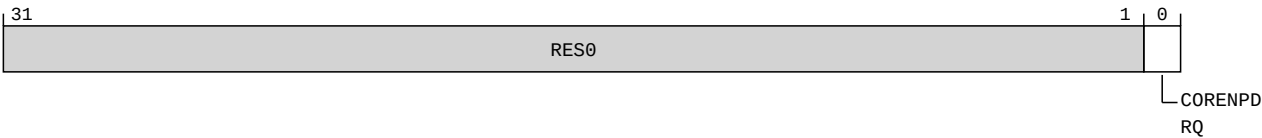
Configuration

AArch32 System register DBGPRCR bit [0] is architecturally mapped to External register [EDPRCR](#)[0].

Attributes

DBGPRCR is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

CORENPD RQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPD RQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of [EDPRCR](#).COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see ‘[Core power domain power states](#)’.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in [EDPRCR](#).COREPURQ.

Accessing DBGPRCR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0100	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDOSA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGPRCR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGPRCR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0100	0b100

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDOSA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGPRCR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    DBGPRCR() = R(t);
end;

```

D10.3.25 DBGVCR, Debug Vector Catch Register

The DBGVCR characteristics are:

Purpose

Controls Vector Catch debug events.

Configuration

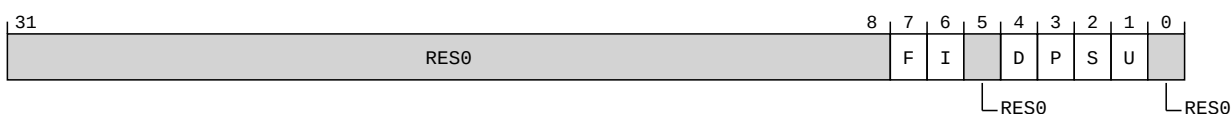
This register is required in all implementations.

Attributes

DBGVCR is a 32-bit register.

Field descriptions

EL3 is not implemented



Bits [31:8]

Reserved, RES0.

F, bit [7]

FIQ vector catch enable.

The exception vector offset is $0 \times 1C$.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [6]

IRQ vector catch enable.

The exception vector offset is 0×18 .

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

D, bit [4]

Data Abort exception vector catch enable.

The exception vector offset is 0×10 .

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P, bit [3]

Prefetch Abort vector catch enable.

The exception vector offset is 0x0C.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [2]

Supervisor Call (SVC) vector catch enable.

The exception vector offset is 0x08.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [1]

Undefined Instruction vector catch enable.

The exception vector offset is 0x04.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing DBGVCR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0111	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGVCR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGVCR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGVCR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    DBGVCR() = R(t);
end;
```

D10.3.26 DBGWCR<n>, Debug Watchpoint Control Registers, n = 0 - 15

The DBGWCR<n> characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register [DBGWVR<n>](#).

Configuration

If watchpoint n is not implemented then accesses to this register are UNDEFINED.

AArch32 System register DBGWCR<n> bits [31:0] are architecturally mapped to External register [DBGWCR<n>_ELI](#)[31:0].

Attributes

DBGWCR<n> is a 32-bit register.

Field descriptions

31	29	28	24	23	21	20	19	16	15	14	13	12	5	4	3	2	1	0
RES0		MASK		RES0	WT		LBN		SSC	HMC		BAS		LSC		PAC		E

When the E field is zero, all the other fields in the register are ignored.

Bits [31:29]

Reserved, RES0.

MASK, bits [28:24]

Address Mask. Only objects up to 2GB can be watched using a single mask.

MASK	Meaning
0b00000	No mask.
0b00011..0b11111	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from 0b00011 masking 3 address bits (0x00000007 mask for address) to 0b11111 masking 31 address bits (0x7FFFFFFF mask for address).

If programmed with a reserved value, the watchpoint behaves as if either:

- DBGWCR<n>.MASK has been programmed with a defined value, which might be 0 (no mask), other than for a direct read of DBGWCR<n>.
- The watchpoint is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [23:21]

Reserved, RES0.

WT, bit [20]

Watchpoint type. Possible values are:

WT	Meaning
0b0	Unlinked data address match.
0b1	Linked data address match.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number. For Linked data address watchpoints, this specifies the index of the context-matching breakpoint linked to.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint *n* is generated. This field must be interpreted along with the HMC and PAC fields.

For more information, see [‘Execution conditions for which a watchpoint generates Watchpoint exceptions’](#), and [‘Reserved DBGWCR<n>. {SSC, HMC, PAC} values’](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint *n* is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see [‘Execution conditions for which a watchpoint generates Watchpoint exceptions’](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or doubleword addressed by [DBGWVR<n>](#) is being watched.

BAS	Description
0bxxxxxxx1	Match byte at DBGWVR<n>
0bxxxxxx1x	Match byte at DBGWVR<n>+1
0bxxxxx1xx	Match byte at DBGWVR<n>+2
0bxxxx1xxx	Match byte at DBGWVR<n>+3

In cases where [DBGWVR<n>](#) addresses a doubleword:

BAS	Description, if DBGWVR<n> [2] == 0
0bxxx1xxxx	Match byte at DBGWVR<n> +4
0bxx1xxxxx	Match byte at DBGWVR<n> +5
0bx1xxxxxx	Match byte at DBGWVR<n> +6
0b1xxxxxxx	Match byte at DBGWVR<n> +7

If [DBGWVR<n>](#)[2] == 1, only BAS[3:0] are used and BAS[7:4] are ignored. Arm deprecates setting [DBGWVR<n>](#)[2] == 1.

The valid values for BAS are nonzero binary numbers all of whose set bits are contiguous. All other values are reserved and must not be used by software. See ‘[Reserved DBGWCR<n>.BAS values](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

LSC	Meaning
0b01	Match instructions that load from a watchpointed address.
0b10	Match instructions that store to a watchpointed address.
0b11	Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see ‘[Execution conditions for which a watchpoint generates Watchpoint exceptions](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable watchpoint n. Possible values are:

E	Meaning
0b0	Watchpoint disabled.
0b1	Watchpoint enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGWCR<n>

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b111

```

let m:integer = UInt(CRm[3:0]);

if m >= NUM_WATCHPOINTS then
    Undefined();
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elseif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWCR(m);
    end;
elseif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWCR(m);
    end;
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b111

```

let m:integer = UInt(CRm[3:0]);

if m >= NUM_WATCHPOINTS then
    Undefined();
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then

```



```
        AArch32_TakeHypTrapException(0x05);
    elsif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSCR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR(m) = R(t);
    end;
end;
```

D10.3.27 DBGW FAR, Debug Watchpoint Fault Address Register

The DBGW FAR characteristics are:

Purpose

Previously returned information about the address of the instruction that accessed a watchpointed address. Is now deprecated and RES0.

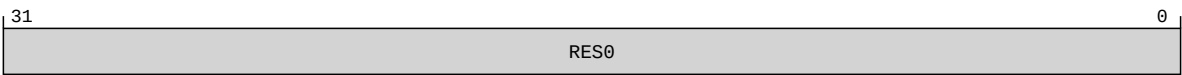
Configuration

There are no configuration notes.

Attributes

DBGW FAR is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing DBGW FAR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        R(t) = DBGW FAR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = DBGW FAR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    else
        DBGWFSR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    DBGWFSR() = R(t);
end;
```

D10.3.28 DBGWVR<n>, Debug Watchpoint Value Registers, n = 0 - 15

The DBGWVR<n> characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>.

Configuration

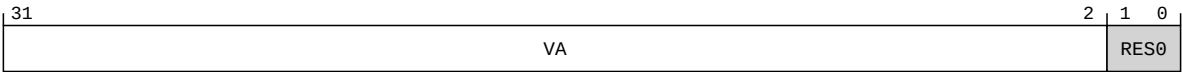
If watchpoint n is not implemented then accesses to this register are UNDEFINED.

AArch32 System register DBGWVR<n> bits [31:0] are architecturally mapped to External register DBGWVR<n>_EL1[31:0].

Attributes

DBGWVR<n> is a 32-bit register.

Field descriptions



VA, bits [31:2]

Bits[31:2] of the address value for comparison.

Arm deprecates setting DBGWVR<n>[2] = 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGWVR<n>

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b110

```
let m:integer = UInt(CRm[3:0]);

if m >= NUM_WATCHPOINTS then
    Undefined();
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
```

```
        AArch32_TakeHypTrapException(0x05);
    elseif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWVR(m);
    end;
elseif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        R(t) = DBGWVR(m);
    end;
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-15

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0000	m[3:0]	0b110

```
let m:integer = UInt(CRm[3:0]);

if m >= NUM_WATCHPOINTS then
    Undefined();
elseif PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().[TDE,TDA] != '00' then
        AArch32_TakeHypTrapException(0x05);
    elseif DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR(m) = R(t);
    end;
elseif PSTATE.EL == EL2 then
    if DBGOSLSR().OSLK == '0' && HaltingAllowed() && EDSR().TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR(m) = R(t);
    end;
end;
```

D10.3.29 DLR, Debug Link Register

The DLR characteristics are:

Purpose

In Debug state, holds the address to restart from.

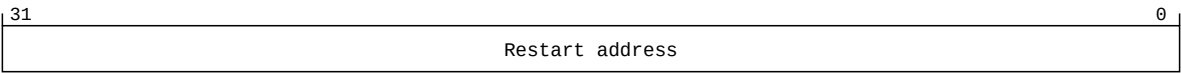
Configuration

This register is a Special-purpose register.

Attributes

DLR is a 32-bit register.

Field descriptions



ADDR, bits [31:0]

Restart address.

Accessing DLR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b001

```
if !Halted() then
    Undefined();
else
    R(t) = DLR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b001

```
if !Halted() then
    Undefined();
else
    DLR() = R(t);
end;
```

D10.3.30 DSPSR, Debug Saved Program Status Register

The DSPSR characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

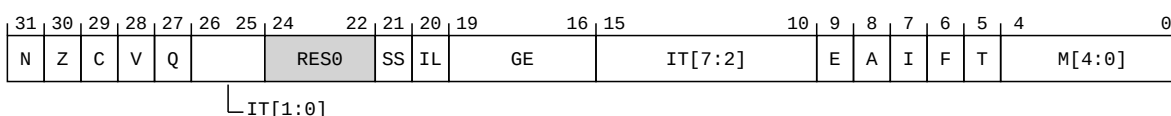
Configuration

This register is a Special-purpose register.

Attributes

DSPSR is a 32-bit register.

Field descriptions



N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on entering Debug state, and copied to PSTATE.Q on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IT, bits [15:10, 26:25]

If-Then. Set to the value of PSTATE.IT on entering Debug state, and copied to PSTATE.IT on exiting Debug state.

On exiting Debug state, DSPSR.IT must contain a value that is valid for the instruction being returned to.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [24:22]

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on entering Debug state, and copied to PSTATE.GE on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on entering Debug state, and copied to PSTATE.E on exiting Debug state.

If the implementation does not support big-endian operation, DSPSR.E is RES0. If the implementation does not support little-endian operation, DSPSR.E is RES1. On exiting Debug state, if the implementation does not support big-endian operation at the Exception level being returned to, DSPSR.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, DSPSR.E is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on entering Debug state, and copied to PSTATE.T on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[4:0], bits [4:0]

Mode. Set to the value of PSTATE.M[4:0] on entering Debug state, and copied to PSTATE.M[4:0] on exiting Debug state.

M[4:0]	Meaning
0b10000	User.
0b10001	FIQ.
0b10010	IRQ.
0b10011	Supervisor.
0b10111	Abort.
0b11011	Undefined.
0b11111	System.

Other values are reserved. If DSPSR.M[4:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in [‘Illegal return events from AArch32 state’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DSPSR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b000

```
if !Halted() then
    Undefined();
else
    R(t) = DSPSR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b011	0b0100	0b0101	0b000

```
if !Halted() then
    Undefined();
else
    DSPSR() = R(t);
end;
```

D10.3.31 HDCCR, Hyp Debug Control Register

The HDCCR characteristics are:

Purpose

Controls the trapping to Hyp mode of accesses, at EL1 or lower, to functions provided by the debug and trace architectures and the Performance Monitors Extension.

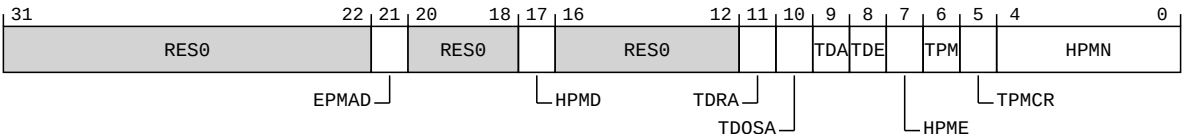
Configuration

There are no configuration notes.

Attributes

HDCCR is a 32-bit register.

Field descriptions



Bits [31:22]

Reserved, RES0.

EPMAD, bit [21]

External debug interface access to Hyp mode Performance Monitors registers disable. This disables access by an external debugger to counters that are reserved for use from Hyp mode.

EPMAD	Meaning
0b0	Access to all Performance Monitors counters by an external debugger is permitted.
0b1	Access to Performance Monitors counters in the range [HDCCR.HPMN..(PMCR.N-1)] by an external debugger is disabled, unless overridden by ExternalHypNoninvasiveDebugEnabled().

If the Performance Monitors Extension is not implemented or does not support external debug interface accesses this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [20:18]

Reserved, RES0.

HPMD, bit [17]

Guest Performance Monitors Disable. Controls PMU operation in Hyp mode.

HPMD	Meaning
0b0	Counters are not affected by this mechanism.
0b1	Affected counters are prohibited from counting in Hyp mode. If PMCR.DP is 1, then PMCCNTR is disabled in Hyp mode. Otherwise, PMCCNTR is not affected by this mechanism.

The counters affected by this field are:

- Event counters [PMEVCNTR<n>](#) for values of n less than [HDCR.HPMN](#).
- If [PMCR.DP](#) is 1, the cycle counter [PMCCNTR](#).

Other event counters are not affected by this field.

When [PMCR.DP](#) is 0, [PMCCNTR](#) is not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [16:12]

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps EL0 and EL1 System register MRC or MCR accesses, reported using EC syndrome value 0x05, and MRRC accesses, reported using EC syndrome value 0x0C, to the Debug ROM registers to Hyp mode.

TDRA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL0 and EL1 System register accesses to the DBGDRAR or DBGDSAR are trapped to Hyp mode, unless it is trapped by DBGDSCRExt.UDCCdis .

If [HCR.TGE](#) or [HDCR.TDE](#) is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TDOSA, bit [10]

Trap debug OS-related register access. Traps EL1 System register MRC or MCR accesses, reported using EC syndrome value 0x05, to the powerdown debug System registers to Hyp mode.

TDOSA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 System register accesses to the powerdown debug System registers are trapped to Hyp mode.

The registers for which accesses are trapped are as follows:

- [DBGOSLSR](#), [DBGOSLAR](#), [DBGOSDLR](#), and [DBGPRCR](#).

- Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

Note

These registers are not accessible at EL0.

If [HCR.TGE](#) or [HDCR.TDE](#) is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TDA, bit [9]

Trap debug access. Traps EL0 and EL1 System register MRC or MCR accesses, reported using EC syndrome value 0x05, to those debug System registers in the (coproc==0b1110) encoding space that are not trapped by either of the following:

- [HDCR.TDRA](#).
- [HDCR.TDOSA](#).

TDA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL0 or EL1 System register accesses to the debug System registers, other than the registers trapped by HDCR.TDRA and HDCR.TDOSA , are trapped to Hyp mode, unless it is trapped by DBGDSCRext.UDCCdis .

Traps of AArch32 accesses to [DBGDTRRXint](#) and [DBGDTRTXint](#) are ignored in Debug state.

If [HCR.TGE](#) or [HDCR.TDE](#) is 1, behavior is as if this bit is 1 other than for the purpose of a direct read.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TDE, bit [8]

Trap Debug exceptions. Controls routing of Debug exceptions, and defines the debug target Exception level, EL_D .

TDE	Meaning
0b0	The debug target Exception level is EL1.
0b1	Debug exceptions generated at EL1 or EL0 are routed to EL2. The HDCR.{TDRA, TDOSA, TDA} fields are treated as being 1 for all purposes other than returning the result of a direct read of the register.

For more information, see '[Routing debug exceptions](#)'.

When [HCR.TGE](#) = 1, the PE behaves as if the value of this field is 1 for all purposes other than returning the value of a direct read of the register.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

HPME, bit [7]

When FEAT_PMUv3 is implemented:

Hyp Enable.

HPME	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET .

The counters affected by this field are event counters [PMEVCNTR<n>](#) for values of n greater than or equal to HDCR.HPMN and less than [PMCR.N](#).

Other event counters and [PMCCNTR](#) are not affected by this field.

If HDCR.HPMN is equal to [PMCR.N](#), then this field has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

TPM, bit [6]

When FEAT_PMUv3 is implemented:

Trap accesses of PMU System registers. Enables a trap to EL2 on accesses of PMU System registers.

TPM	Meaning
0b0	Accesses of the specified PMU System registers are not trapped by this mechanism.
0b1	Accesses of the specified PMU System registers at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.

The instructions affected by this control are:

- MRC and MCR accesses to [PMCCFILTR](#), [PMCCNTR](#), [PMCNTENCLR](#), [PMCNTENSET](#), [PMCR](#), [PMEVCNTR<n>](#), [PMEVTYPER<n>](#), [PMINTENCLR](#), [PMINTENSET](#), [PMOVSRR](#), [PMOVSSET](#), [PMSELR](#), [PMSWINC](#), [PMUSERENR](#), [PMXVCNTR](#), and [PMXEVTYPER](#).
- MRC accesses to [PMCEID0](#) and [PMCEID1](#).
- MRRC and MCRR accesses to [PMCCNTR](#).

Unless the instruction generates a higher priority exception, trapped instructions generate a Hyp Trap exception.

Trapped instructions are reported using EC syndrome value 0x03 for MRC and MCR accesses, and 0x04 for MRRC and MCRR accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

TPMCR, bit [5]

When FEAT_PMUv3 is implemented:

Trap [PMCR](#) accesses. Traps EL0 and EL1 MCR or MRC accesses to the [PMCR](#) to Hyp mode, reported using EC syndrome value 0x03.

TPMCR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL0 and EL1 accesses to the PMCR are trapped to Hyp mode, unless it is trapped by PMUSERENR.EN .

Note

EL2 does not provide traps on Performance Monitor System register accesses through the optional memory-mapped external debug interface.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

HPMN, bits [4:0]

When FEAT_PMUv3 is implemented:

Defines the number of event counters [PMEVCNTR<n>](#) that are accessible from EL1 and, if permitted, from EL0.

HDCR.HPMN divides the event counters into a first range and a second range.

If HDCR.HPMN is not 0 and is less than the number of PMU event counters implemented by the PE, `NUM_PMU_COUNTERS`, then event counters `[0..(HDCR.HPMN-1)]` are in the first range, and the remaining event counters `[HDCR.HPMN..(NUM_PMU_COUNTERS-1)]` are in the second range.

If HDCR.HPMN is equal to `NUM_PMU_COUNTERS`, or EL2 is not implemented, then all of the following apply:

- All event counters are in the first range.
- No event counters are in the second range.

All of the following apply for an event counter [PMEVCNTR<n>](#) in the first range:

- The counter is accessible from EL1 and EL2.
- The counter is accessible from EL0 if permitted by [PMUSERENR](#).
- [PMCR.E](#) and [PMCNTENSET\[n\]](#) enable the operation of the event counter.

All of the following apply for an event counter [PMEVCNTR<n>](#) in the second range:

- The counter is only accessible from EL2.
- HDCR.HPME and [PMCNTENSET\[n\]](#) enable the operation of the event counter.

Values greater than `NUM_PMU_COUNTERS` and 0 are reserved.

If this field is set to a reserved value, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- The number of event counters in each of the first and second ranges is UNKNOWN. That is, either:
 - The PE behaves as if HDCR.HPMN is set to an UNKNOWN nonzero value less than or equal to `NUM_PMU_COUNTERS`.
 - All counters are in the second range and none are in the first range.

The reset behavior of this field is:

- On a Warm reset, this field resets to the expression `NUM_PMU_COUNTERS`.

Otherwise:

Reserved, `RES0`.

Accessing HDCR

Accesses to this register use the following encodings in the System register encoding space:

`MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}`

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = HDCR();
end;
```

`MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}`

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T1 == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        Undefined();
    end;
elsif PSTATE.EL == EL2 then
    HDCR() = R(t);
end;
```


D10.4 Performance Monitors registers

This section lists the Performance Monitors registers in AArch32.

D10.4.1 PMCCFILTR, Performance Monitors Cycle Count Filter Register

The PMCCFILTR characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR, increments.

Configuration

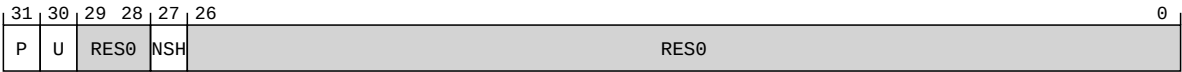
AArch32 System register PMCCFILTR bits [31:0] are architecturally mapped to External register PMCCFILTR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR are UNDEFINED.

Attributes

PMCCFILTR is a 32-bit register.

Field descriptions



P, bit [31]

Privileged filtering. Controls counting cycles in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL1.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL1 is further controlled by PMCCFILTR.NSK.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

U, bit [30]

User filtering. Controls counting cycles in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL0.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL0 is further controlled by PMCCFILTR.NSU.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 filtering. Controls counting cycles in EL2.

NSH	Meaning
0b0	The PE does not count cycles in EL2.
0b1	This mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [26:0]

Reserved, RES0.

Accessing PMCCFILTR

PMCCFILTR can also be accessed by using [PMXEVTYPER](#) with [PMSELR.SEL](#) set to 0b11111.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b1111	0b111

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCCFILTR();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCCFILTR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMCCFILTR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b1111	0b111

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCCFILTR() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCCFILTR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMCCFILTR() = R(t);
end;

```

D10.4.2 PMCCNTR, Performance Monitors Cycle Count Register

The PMCCNTR characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See ‘Time as measured by the Performance Monitors cycle counter’ for more information.

[PMCCFILTR](#) determines the modes and states in which the PMCCNTR can increment.

Configuration

PMCCNTR is a 64-bit register that can also be accessed as a 32-bit value. If it is accessed as a 32-bit register, accesses read and write bits [31:0] and do not modify bits [63:32].

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is **CONSTRAINED UNPREDICTABLE** whether or not PMCCNTR continues to increment when clocks are stopped by WFI and WFE instructions.

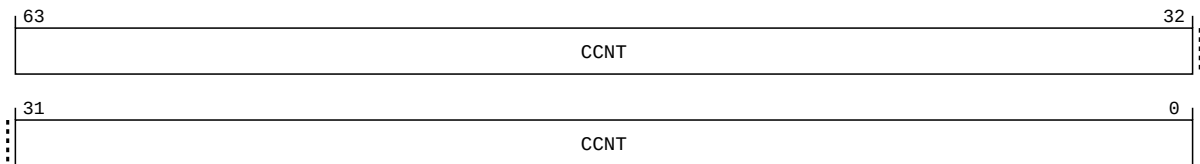
AArch32 System register PMCCNTR bits [63:0] are architecturally mapped to External register [PMCCNTR_EL0](#)[63:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR are **UNDEFINED**.

Attributes

PMCCNTR is a 64-bit register.

Field descriptions



CCNT, bits [63:0]

Cycle count. Depending on the values of [PMCR](#).{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to [PMCR](#).C sets this field to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Accessing PMCCNTR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b000

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().[CR,EN] == '00' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCCNTR()[31:0];
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCCNTR()[31:0];
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMCCNTR()[31:0];
end;

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

```

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b000

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCCNTR()[31:0] = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else

```

```
PMCCNTR() [31:0] = R(t);
end;
elsif PSTATE.EL == EL2 then
    PMCCNTR() [31:0] = R(t);
end;
```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1001	0b0000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().[CR,EN] == '00' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x04);
    else
        R(t, t2) = PMCCNTR();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x04);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x04);
    else
        R(t, t2) = PMCCNTR();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = PMCCNTR();
end;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1001	0b0000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
```

```
        AArch32_TakeHypTrapException(0x04);
    elseif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x04);
    else
        PMCCNTR() = R(t, t2);
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x04);
    elseif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x04);
    else
        PMCCNTR() = R(t, t2);
    end;
elseif PSTATE.EL == EL2 then
    PMCCNTR() = R(t, t2);
end;
```

D10.4.3 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x0000 to 0x001F.

For more information about the Common events and the use of the PMCEIDn registers, see ‘The PMU event number space and common events’.

Configuration

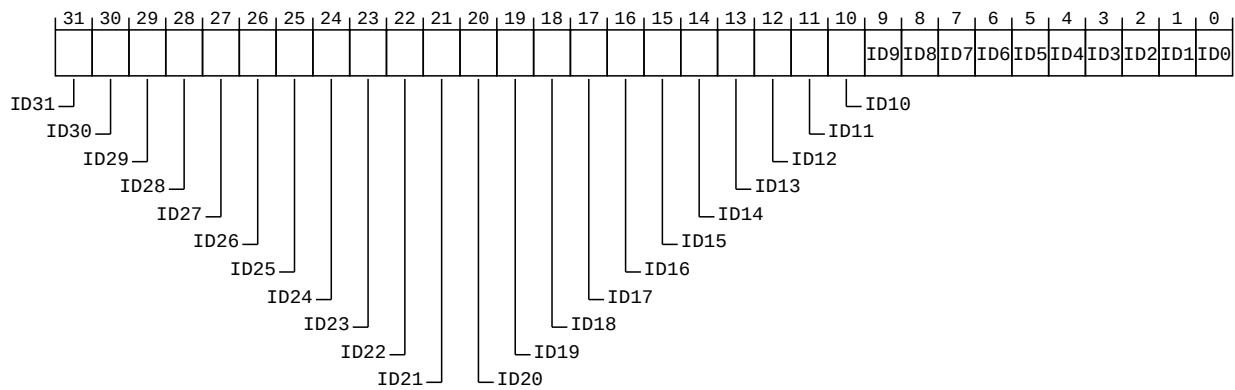
AArch32 System register PMCEID0 bits [31:0] are architecturally mapped to External register PMCEID0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID0 are UNDEFINED.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions



ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

ID<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID0

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b110

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCEID0();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCEID0();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMCEID0();
end;

```

D10.4.4 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x0020 to 0x003F.

For more information about the Common events and the use of the PMCEIDn registers see ‘[The PMU event number space and common events](#)’.

Configuration

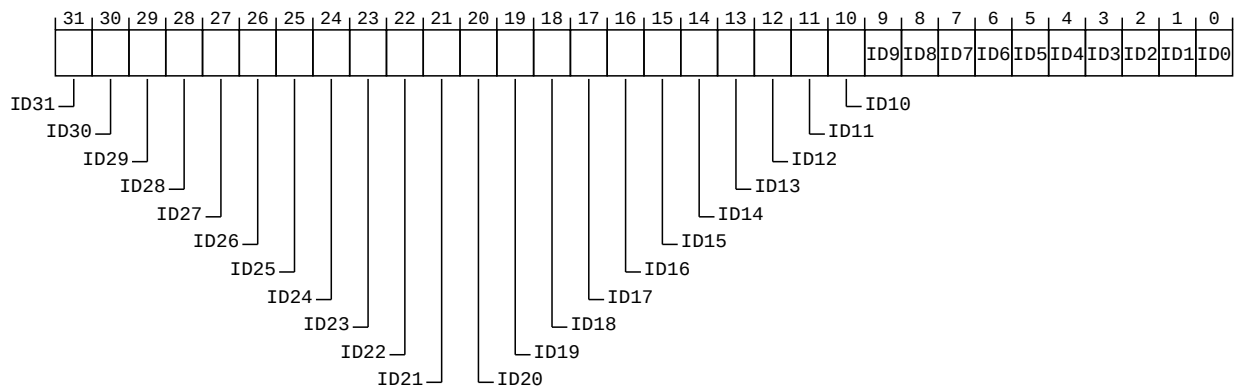
AArch32 System register PMCEID1 bits [31:0] are architecturally mapped to External register [PMCEID1](#)[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID1 are UNDEFINED.

Attributes

PMCEID1 is a 32-bit register.

Field descriptions



ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event (0x0020 + n).

For each bit:

ID<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID1

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b111

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCEID1();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCEID1();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMCEID1();
end;

```

D10.4.5 PMCNTENCLR, Performance Monitors Count Enable Clear register

The PMCNTENCLR characteristics are:

Purpose

Allows software to disable the following counters:

- The cycle counter [PMCCNTR](#).
- The event counters [PMEVCNTR<n>](#).

Reading from this register shows which counters are enabled.

Configuration

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to AArch32 System register [PMCNTENSET](#)[31:0].

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to External register [PMCNTENCLR_ELO](#)[31:0].

AArch32 System register PMCNTENCLR bits [31:0] are architecturally mapped to External register [PMCNTENSET_ELO](#)[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR are UNDEFINED.

Attributes

PMCNTENCLR is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

[PMCCNTR](#) disable. On writes, allows software to disable [PMCCNTR](#). On reads, returns the [PMCCNTR](#) enable status.

C	Meaning
0b0	PMCCNTR disabled.
0b1	PMCCNTR enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

[PMEVCNTR<m>](#) disable. On writes, allows software to disable [PMEVCNTR<m>](#). On reads, returns the [PMEVCNTR<m>](#) enable status.

P<m>	Meaning
0b0	PMEVCNTR<m> disabled.
0b1	PMEVCNTR<m> enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{GetNumEventCountersAccessible}()$, access to this field is RAZ/WI.
- Otherwise, access to this field is WIC.

Accessing PMCNTECLR

Accesses to this register use the following encodings in the System register encoding space:

$\text{MRC}\{\langle c \rangle\}\{\langle q \rangle\} \langle \text{coproc} \rangle, \{\#\}\langle \text{opc1} \rangle, \langle \text{Rt} \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle\{\{, \{\#\}\langle \text{opc2} \rangle\}$

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b010

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCNTECLR();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCNTECLR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMCNTECLR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b010

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCNTENCLR() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCNTENCLR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMCNTENCLR() = R(t);
end;

```

D10.4.6 PMCNTENSET, Performance Monitors Count Enable Set register

The PMCNTENSET characteristics are:

Purpose

Allows software to enable the following counters:

- The cycle counter [PMCCNTR](#).
- The event counters [PMEVCNTR<n>](#).

Reading from this register shows which counters are enabled.

Configuration

AArch32 System register PMCNTENSET bits [31:0] are architecturally mapped to AArch32 System register [PMCNTENCLR](#)[31:0].

AArch32 System register PMCNTENSET bits [31:0] are architecturally mapped to External register [PMCNTENSET_ELO](#)[31:0].

AArch32 System register PMCNTENSET bits [31:0] are architecturally mapped to External register [PMCNTENCLR_ELO](#)[31:0].

This register is present only when FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMCNTENSET are UNDEFINED.

Attributes

PMCNTENSET is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

[PMCCNTR](#) enable. On writes, allows software to enable [PMCCNTR](#). On reads, returns the [PMCCNTR](#) enable status.

C	Meaning
0b0	PMCCNTR disabled.
0b1	PMCCNTR enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

[PMEVCNTR<m>](#) enable. On writes, allows software to enable [PMEVCNTR<m>](#). On reads, returns the [PMEVCNTR<m>](#) enable status.

P<m>	Meaning
0b0	PMEVCNTR<m> disabled.
0b1	PMEVCNTR<m> enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{GetNumEventCountersAccessible}()$, access to this field is RAZ/WI.
- Otherwise, access to this field is WIS.

Accessing PMCNTENSET

Accesses to this register use the following encodings in the System register encoding space:

$\text{MRC}\{\langle c \rangle\}\{\langle q \rangle\} \langle \text{coproc} \rangle, \{\#\}\langle \text{opc1} \rangle, \langle \text{Rt} \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle\{, \{\#\}\langle \text{opc2} \rangle\}$

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b001

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCNTENSET();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCNTENSET();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMCNTENSET();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b001

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCNTENSET() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCNTENSET() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMCNTENSET() = R(t);
end;

```

D10.4.7 PMCR, Performance Monitors Control Register

The PMCR characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configuration

AArch32 System register PMCR bits [10:0] are architecturally mapped to External register [PMCR_EL0](#)[10:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCR are UNDEFINED.

Attributes

PMCR is a 32-bit register.

Field descriptions

31	24	23	16	15	11	10	7	6	5	4	3	2	1	0	
IMP		IDCODE			N		RES0		LC	DP	X	D	C	P	E

IMP, bits [31:24]

Implementer code.

If this field is zero, then PMCR.IDCODE is RES0 and software must use [MIDR](#) to identify the PE.

Otherwise, this field and PMCR.IDCODE identify the PMU implementation to software. The implementer codes are allocated by Arm. A nonzero value has the same interpretation as [MIDR](#).Implementer.

Arm deprecates use of this field.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

IDCODE, bits [23:16]

When PMCR.IMP != '00000000':

Identification code. Arm deprecates use of this field.

Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

N, bits [15:11]

Indicates the number of event counters implemented. This value is in the range of 0b000000-0b111111. If the value is 0b000000, then only [PMCCNTR](#) is implemented. If the value is 0b111111, then [PMCCNTR](#) and 31 event counters are implemented.

If EL2 is implemented, then all of the following apply:

- If EL2 is using AArch32, then reads of this field from Non-secure EL1 and Non-secure EL0 return the Effective value of [HDCR](#).HPMN.
- If EL2 is enabled in the current Security state and using AArch64, then reads of this field from EL1 and EL0 return the Effective value of [MDCR_EL2](#).HPMN.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [10:7]

Reserved, RES0.

LC, bit [6]

Long cycle counter enable. Determines when unsigned overflow is recorded by [PMOVS](#).C.

LC	Meaning
0b0	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR [31:0].
0b1	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR [63:0].

Arm deprecates use of [PMCR](#).LC = 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DP, bit [5]

Disable cycle counter when event counting is prohibited.

DP	Meaning
0b0	Cycle counting by PMCCNTR is not affected by this mechanism.
0b1	Cycle counting by PMCCNTR is disabled in prohibited regions:

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited. For more information about event counter ranges, see [HDCR](#).HPMN.

For more information, see ‘[Prohibiting event and cycle counting](#)’.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
0b0	Do not export events.
0b1	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If [FEAT_ETMv4](#) is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

Clock divider.

D	Meaning
0b0	When enabled, PMCCNTR counts every clock cycle.
0b1	When enabled, PMCCNTR counts once every 64 clock cycles.

If the Effective value of PMCR.LC is 1, then this field is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR.D = 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
0b0	No action.
0b1	Reset PMCCNTR to zero.

Note

Resetting [PMCCNTR](#) does not change the cycle counter overflow field. The value of PMCR.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

P	Meaning
0b0	No action.
0b1	Reset all affected event counters PMEVCNTR<n> to zero.

The event counters affected by this field are:

- All event counters in the first range.
- If any of the following are true, all event counters in the second range:
 - EL2 is not implemented.
 - The PE is executing at EL2.

Writes to this field do not affect other event counters or the cycle counter [PMCCNTR](#).

For more information about event counter ranges, see [HDCR.HPMN](#).

Note

Resetting the event counters does not change the event counter overflow fields.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET .

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see [HDCR.HPMN](#).
- The cycle counter [PMCCNTR](#).

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PMCR

Accesses to this register use the following encodings in the System register encoding space:

`MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}`

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b000

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPMCR == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCR();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPMCR == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMCR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMCR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b000

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPMCR == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCR() = R(t);
    end;

```

```
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && HDCR().TPMCR == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMCR() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    PMCR() = R(t);
end;
```

D10.4.8 PMEVCNTR<n>, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n> characteristics are:

Purpose

Holds event counter n, which counts events, where n is 0 to 30.

Configuration

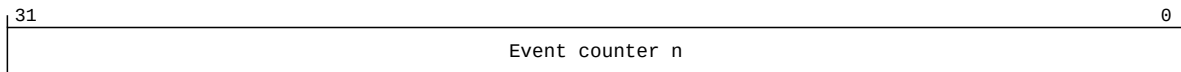
AArch32 System register PMEVCNTR<n> bits [31:0] are architecturally mapped to External register [PMEVCNTR<n>_EL0](#)[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n> are UNDEFINED.

Attributes

PMEVCNTR<n> is a 32-bit register.

Field descriptions



EVCNT, bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTR<n>

PMEVCNTR<n> can also be accessed by using [PMXEVCNTR](#) with [PMSELR](#).SEL set to n.

If <n> is greater than or equal to the number of accessible event counters, then reads and writes of [PMEVCNTR<n>](#) are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by [PMUSERENR](#).{ER,EN} or [PMUSERENR_EL0](#).{ER,EN}.

At EL0 and EL1, [HDCR](#).HPMN identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see [HDCR](#).HPMN.

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-30

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b10:m[4:3]	m[2:0]

```

let m:integer = UInt(CRm[1:0] :: opc2[2:0]);

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR().[ER,EN] == '00' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        R(t) = PMEVCNTR(m);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        R(t) = PMEVCNTR(m);
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMEVCNTR(m);
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-30

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b10:m[4:3]	m[2:0]

```

let m:integer = UInt(CRm[1:0] :: opc2[2:0]);

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    end;

```

```
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVCNTR(m) = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVCNTR(m) = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMEVCNTR(m) = R(t);
end;
```

D10.4.9 PMEVTYPER<n>, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n> characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configuration

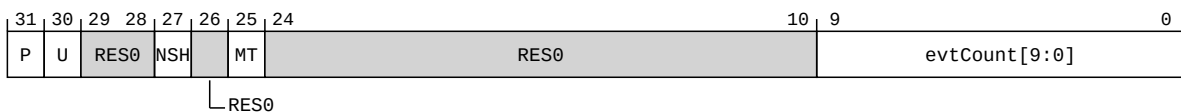
AArch32 System register PMEVTYPER<n> bits [31:0] are architecturally mapped to External register PMEVTYPER<n>_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPEPER<n> are UNDEFINED.

Attributes

PMEVTYPEP<n> is a 32-bit register.

Field descriptions



P, bit [31]

Privileged filtering. Controls counting events in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL1.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL1 is further controlled by PMEVTYPEPER<n>.NSK.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

User filtering. Controls counting events in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL0.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL0 is further controlled by PMEVTYPER<n>.NSU.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 filtering. Controls counting events in EL2.

NSH	Meaning
0b0	The PE does not count events in EL2.
0b1	This mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [26]

Reserved, RES0.

MT, bit [25]

When an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

MT	Meaning
0b0	Count events only on controlling PE.
0b1	Count events from any PE with the same affinity at level 1 and above as this PE.

Unless otherwise stated:

- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycles when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event.

The second condition is any condition in addition to this.

For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache.

For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

See [‘Multithreaded implementations’](#) and [‘Cycle event counting in multithreaded implementations’](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:10]

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter [PMEVCNTR<n>](#).

The ranges of event numbers allocated to each type of event are shown in ‘[Allocation of the PMU event number space](#)’.

If [PMEVTYPER<n>](#).evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range 0x0000 to 0x003F, no events are counted and the value returned by a direct or external read of the [PMEVTYPER<n>](#).evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the [PMEVTYPER<n>](#).evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<n>

[PMEVTYPER<n>](#) can also be accessed by using [PMXEVTYPER](#) with [PMSELR.SEL](#) set to n.

If <n> is greater than or equal to the number of accessible event counters, then reads and writes of [PMEVTYPER<n>](#) are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by [PMUSERENR.EN](#) or [PMUSERENR_EL0.EN](#).

At EL0 and EL1 [HDCR.HPMN](#) identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see [HDCR.HPMN](#).

Accesses to this register use the following encodings in the System register encoding space:

`MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}` ; Where m = 0-30

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b11:m[4:3]	m[2:0]

```

let m:integer = UInt(CRm[1:0] :: opc2[2:0]);

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        R(t) = PMEVTYPER(m);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        R(t) = PMEVTYPER(m);
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMEVTYPER(m);
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>} ; Where m = 0-30

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b11:m[4:3]	m[2:0]

```

let m:integer = UInt(CRm[1:0] :: opc2[2:0]);

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif m >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else

```

```
        PMEVTYPER(m) = R(t);
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elseif EL2Enabled() && m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVTYPER(m) = R(t);
    end;
elseif PSTATE.EL == EL2 then
    PMEVTYPER(m) = R(t);
end;
```

D10.4.10 PMINTENCLR, Performance Monitors Interrupt Enable Clear register

The PMINTENCLR characteristics are:

Purpose

Allows software to disable the generation of interrupt requests on overflows from the following counters:

- The cycle counter [PMCCNTR](#).
- The event counters [PMEVCNTR<n>](#).

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to AArch32 System register [PMINTENSET](#)[31:0].

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to External register [PMINTENCLR_EL1](#)[31:0].

AArch32 System register PMINTENCLR bits [31:0] are architecturally mapped to External register [PMINTENSET_EL1](#)[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENCLR are UNDEFINED.

Attributes

PMINTENCLR is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request on unsigned overflow of [PMCCNTR](#) disable. On writes, allows software to disable the interrupt request on unsigned overflow of [PMCCNTR](#). On reads, returns the interrupt request on unsigned overflow of [PMCCNTR](#) enable status.

C	Meaning
0b0	Interrupt request on unsigned overflow of PMCCNTR disabled.
0b1	Interrupt request on unsigned overflow of PMCCNTR enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

P<m>, bits [m], for m = 30 to 0

Interrupt request on unsigned overflow of [PMEVCNTR<m>](#) disable. On writes, allows software to disable the interrupt request on unsigned overflow of [PMEVCNTR<m>](#). On reads, returns the interrupt request on unsigned overflow of [PMEVCNTR<m>](#) enable status.

P<m>	Meaning
0b0	Interrupt request on unsigned overflow of PMEVCNTR<m> disabled.
0b1	Interrupt request on unsigned overflow of PMEVCNTR<m> enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{GetNumEventCountersAccessible}()$, access to this field is RAZ/WI.
- Otherwise, access to this field is WIC.

Accessing PMINTENCLR

Accesses to this register use the following encodings in the System register encoding space:

$\text{MRC}\{\langle c \rangle\}\{\langle q \rangle\} \langle \text{coproc} \rangle, \{\#\}\langle \text{opc1} \rangle, \langle \text{Rt} \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle\{, \{\#\}\langle \text{opc2} \rangle\}$

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b010

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMINTENCLR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMINTENCLR();
end;

```

$\text{MCR}\{\langle c \rangle\}\{\langle q \rangle\} \langle \text{coproc} \rangle, \{\#\}\langle \text{opc1} \rangle, \langle \text{Rt} \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle\{, \{\#\}\langle \text{opc2} \rangle\}$

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b010

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then

```

```
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMINTENCLR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMINTENCLR() = R(t);
end;
```

D10.4.11 PMINTENSET, Performance Monitors Interrupt Enable Set register

The PMINTENSET characteristics are:

Purpose

Allows software to enable the generation of interrupt requests on overflows from the following counters:

- The cycle counter [PMCCNTR](#).
- The event counters [PMEVCNTR<n>](#).

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to AArch32 System register [PMINTENCLR](#)[31:0].

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to External register [PMINTENCLR_EL1](#)[31:0].

AArch32 System register PMINTENSET bits [31:0] are architecturally mapped to External register [PMINTENSET_EL1](#)[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET are UNDEFINED.

Attributes

PMINTENSET is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request on unsigned overflow of [PMCCNTR](#) enable. On writes, allows software to enable the interrupt request on unsigned overflow of [PMCCNTR](#). On reads, returns the interrupt request on unsigned overflow of [PMCCNTR](#) enable status.

C	Meaning
0b0	Interrupt request on unsigned overflow of PMCCNTR disabled.
0b1	Interrupt request on unsigned overflow of PMCCNTR enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is WIS.

P<m> , bits [m], for m = 30 to 0

Interrupt request on unsigned overflow of [PMEVCNTR<m>](#) enable. On writes, allows software to enable the interrupt request on unsigned overflow of [PMEVCNTR<m>](#). On reads, returns the interrupt request on unsigned overflow of [PMEVCNTR<m>](#) enable status.

P<m>	Meaning
0b0	Interrupt request on unsigned overflow of PMEVCNTR<m> disabled.
0b1	Interrupt request on unsigned overflow of PMEVCNTR<m> enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{GetNumEventCountersAccessible}()$, access to this field is RAZ/WI.
- Otherwise, access to this field is WIS.

Accessing PMINTENSET

Accesses to this register use the following encodings in the System register encoding space:

$\text{MRC}\{\langle c \rangle\}\{\langle q \rangle\} \langle \text{coproc} \rangle, \{\#\}\langle \text{opc1} \rangle, \langle \text{Rt} \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle\{, \{\#\}\langle \text{opc2} \rangle\}$

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b001

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMINTENSET();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMINTENSET();
end;

```

$\text{MCR}\{\langle c \rangle\}\{\langle q \rangle\} \langle \text{coproc} \rangle, \{\#\}\langle \text{opc1} \rangle, \langle \text{Rt} \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle\{, \{\#\}\langle \text{opc2} \rangle\}$

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b001

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then

```

```
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMINTENSET() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMINTENSET() = R(t);
end;
```

D10.4.12 PMOVSr, Performance Monitors Overflow Flag Status Register

The PMOVSr characteristics are:

Purpose

Allows software to clear the unsigned overflow flags for the following counters to 0:

- The cycle counter [PMCCNTR](#).
- The event counters [PMEVCNTR<n>](#).

Reading from this register shows the current unsigned overflow flag values.

Configuration

AArch32 System register PMOVSr bits [31:0] are architecturally mapped to AArch32 System register [PMOVSSET](#)[31:0].

AArch32 System register PMOVSr bits [31:0] are architecturally mapped to External register [PMOVSCLR_EL0](#)[31:0].

AArch32 System register PMOVSr bits [31:0] are architecturally mapped to External register [PMOVSSET_EL0](#)[31:0].

This register is present only when FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMOVSr are UNDEFINED.

Attributes

PMOVSr is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for [PMCCNTR](#) clear. On writes, allows software to clear the unsigned overflow flag for [PMCCNTR](#) to 0. On reads, returns the unsigned overflow flag for [PMCCNTR](#) overflow status.

C	Meaning
0b0	PMCCNTR has not overflowed.
0b1	PMCCNTR has overflowed.

[PMCR.LC](#) controls whether an overflow is detected from unsigned overflow of [PMCCNTR](#)[31:0] or unsigned overflow of [PMCCNTR](#)[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for **PMEVCNTR<m>** clear. On writes, allows software to clear the unsigned overflow flag for **PMEVCNTR<m>** to 0. On reads, returns the unsigned overflow flag for **PMEVCNTR<m>** overflow status.

P<m>	Meaning
0b0	PMEVCNTR<m> has not overflowed.
0b1	PMEVCNTR<m> has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{GetNumEventCountersAccessible}()$, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing PMOVSr

Accesses to this register use the following encodings in the System register encoding space:

$\text{MRC}\{\langle c \rangle\}\{\langle q \rangle\} \langle \text{coproc} \rangle, \{\#\}\langle \text{opc1} \rangle, \langle \text{Rt} \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle\{, \{\#\}\langle \text{opc2} \rangle\}$

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b011

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMOVSr();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMOVSr();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMOVSr();
end;

```


MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b011

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMOVSr() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMOVSr() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMOVSr() = R(t);
end;

```

D10.4.13 PMOVSET, Performance Monitors Overflow Flag Status Set register

The PMOVSET characteristics are:

Purpose

Allows software to set the unsigned overflow flags for the following counters to 1:

- The cycle counter [PMCCNTR](#).
- The event counters [PMEVCNTR<n>](#).

Reading from this register shows the current unsigned overflow flag values.

Configuration

AArch32 System register PMOVSET bits [31:0] are architecturally mapped to AArch32 System register [PMOVSRR](#)[31:0].

AArch32 System register PMOVSET bits [31:0] are architecturally mapped to External register [PMOVSSET_EL0](#)[31:0].

AArch32 System register PMOVSET bits [31:0] are architecturally mapped to External register [PMOVSCLR_EL0](#)[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSET are UNDEFINED.

Attributes

PMOVSET is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for [PMCCNTR](#) set. On writes, allows software to set the unsigned overflow flag for [PMCCNTR](#) to 1. On reads, returns the unsigned overflow flag for [PMCCNTR](#) overflow status.

C	Meaning
0b0	PMCCNTR has not overflowed.
0b1	PMCCNTR has overflowed.

[PMCR.LC](#) controls whether an overflow is detected from unsigned overflow of [PMCCNTR](#)[31:0] or unsigned overflow of [PMCCNTR](#)[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for **PMEVCNTR<m>** set. On writes, allows software to set the unsigned overflow flag for **PMEVCNTR<m>** to 1. On reads, returns the unsigned overflow flag for **PMEVCNTR<m>** overflow status.

P<m>	Meaning
0b0	PMEVCNTR<m> has not overflowed.
0b1	PMEVCNTR<m> has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{GetNumEventCountersAccessible}()$, access to this field is RAZ/WI.
- Otherwise, access to this field is WIS.

Accessing PMOVSSET

Accesses to this register use the following encodings in the System register encoding space:

$\text{MRC}\{\langle c \rangle\}\{\langle q \rangle\} \langle \text{coproc} \rangle, \{ \# \} \langle \text{opc1} \rangle, \langle \text{Rt} \rangle, \langle \text{CRn} \rangle, \langle \text{CRm} \rangle \{, \{ \# \} \langle \text{opc2} \rangle \}$

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b011

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMOVSSET();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMOVSSET();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMOVSSET();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b011

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMOVSSET() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMOVSSET() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMOVSSET() = R(t);
end;

```

D10.4.14 PMSELR, Performance Monitors Event Counter Selection Register

The PMSELR characteristics are:

Purpose

- Selects the current event counter [PMEVCNTR<n>](#) or the cycle counter [PMCCNTR](#).
- Used in conjunction with [PMXEVTYPER](#) to determine the event that increments a selected counter, and the modes and states in which the selected counter increments.
- Used in conjunction with [PMXEVCNTR](#) to determine the value of a selected counter.

Configuration

This register is present only when FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMSELR are UNDEFINED.

Attributes

PMSELR is a 32-bit register.

Field descriptions



Bits [31:5]

Reserved, RES0.

SEL, bits [4:0]

Event counter select. Selects the counter accessed by subsequent accesses to [PMXEVTYPER](#) and [PMXEVCNTR](#).

SEL	Meaning
0b00000..0b11110	Select event counter PMEVCNTR<n> , where n is the value of this field: - MRC and MCR of PMXEVTYPER access PMEVTYPER<n>. - MRC and MCR of PMXEVCNTR access PMEVCNTR<n>.
0b11111	Select the cycle counter, PMCCNTR : - MRC and MCR of PMXEVTYPER access PMCCFILTR. - MRC and MCR of PMXEVCNTR are CONSTRAINED UNPREDICTABLE. For more information, see PMXEVCNTR.

For more information about the results of accesses to the event counters, including when PMSELR.SEL is set to the index of an unimplemented or inaccessible event counter, see [PMXEVTYPER](#) and [PMXEVCNTR](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMSELR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b101

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().[ER,EN] == '00' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMSELR();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMSELR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMSELR();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b101

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().[ER,EN] == '00' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMSELR() = R(t);
    end;
elsif PSTATE.EL == EL1 then

```

```
if EL2Enabled() && HSTR().T9 == '1' then
    AArch32_TakeHypTrapException(0x03);
elsif EL2Enabled() && HDCR().TPM == '1' then
    AArch32_TakeHypTrapException(0x03);
else
    PMSELR() = R(t);
end;
elsif PSTATE.EL == EL2 then
    PMSELR() = R(t);
end;
```

D10.4.15 PMSWINC, Performance Monitors Software Increment register

The PMSWINC characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event 0x00. For more information, see ‘SW_INCR’.

Configuration

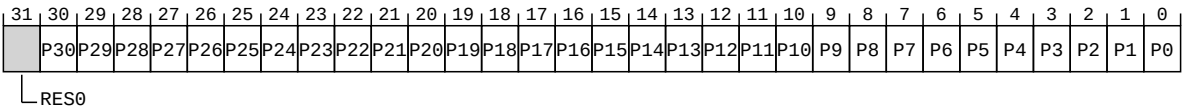
AArch32 System register PMSWINC bits [31:0] are architecturally mapped to External register PMSWINC_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSWINC are UNDEFINED.

Attributes

PMSWINC is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES0.

P<m> , bits [m], for m = 30 to 0

Software increment.

P<m>	Meaning
0b0	Write is ignored.
0b1	Increment PMEVCNTR<m>, if PMEVCNTR<m> is configured to count software increment events.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Accessing PMSWINC

Accesses to this register use the following encodings in the System register encoding space:

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1100	0b100

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if PMUSERENR().[SW,EN] == '00' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMSWINC() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMSWINC() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMSWINC() = R(t);
end;

```

D10.4.16 PMUSERENR, Performance Monitors User Enable Register

The PMUSERENR characteristics are:

Purpose

Enables or disables EL0 access to the Performance Monitors.

Configuration

This register is present only when FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMUSERENR are UNDEFINED.

Attributes

PMUSERENR is a 32-bit register.

Field descriptions

**Bits [31:4]**

Reserved, RES0.

ER, bit [3]

Event counters Read enable.

When PMUSERENR.EN is 0, PMUSERENR.ER enables EL0 reads of the event counters and EL0 reads and writes of the select register.

ER	Meaning
0b0	EL0 reads of the event counters and EL0 reads and writes of the select register are disabled, unless enabled by PMUSERENR.EN.
0b1	EL0 reads of the event counters and EL0 reads and writes of the select register are enabled, unless trapped by another control.

The register accesses affected by this control are:

- MRC reads of **PMEVCNTR<n>** and **PMXEVCNTR**.
- MRC and MCR accesses to **PMSELR**.

When disabled, reads and writes are UNDEFINED.

This field is ignored by the PE when PMUSERENR.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CR, bit [2]

Cycle counter Read enable.

When PMUSERENR.EN is 0, PMUSERENR.CR enables EL0 reads of the cycle counter.

CR	Meaning
0b0	EL0 reads of the cycle counter are disabled, unless enabled by PMUSERENR.EN.
0b1	EL0 reads of the cycle counter are enabled, unless trapped by another control.

The register accesses affected by this control are:

- MRC reads of [PMCCNTR](#).
- MRRC reads of [PMCCNTR](#).

When disabled, reads are UNDEFINED.

This field is ignored by the PE when PMUSERENR.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SW, bit [1]

Software increment register Write enable.

When PMUSERENR.EN is 0, PMUSERENR.SW enables EL0 writes to the Software increment register.

SW	Meaning
0b0	EL0 writes to the Software increment register are disabled, unless enabled by PMUSERENR.EN.
0b1	EL0 writes to the Software increment register are enabled, unless trapped by another control.

The register accesses affected by this control are:

- MCR writes to [PMSWINC](#).

When disabled, writes are UNDEFINED.

This field is ignored by the PE when PMUSERENR.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Enable. Enables EL0 read/write access to PMU System registers.

EN	Meaning
0b0	EL0 accesses to the specified PMU System registers are trapped, unless enabled by PMUSERENR.{ER,CR,SW}.
0b1	EL0 accesses to the specified PMU System registers are enabled, unless trapped by another control.

The register accesses affected by this control are:

- MRC or MCR accesses to [PMCCFILTR](#), [PMCCNTR](#), [PMCNTENCLR](#), [PMCNTENSET](#), [PMCR](#), [PMEVCNTR<n>](#), [PMEVTYPER<n>](#), [PMOVSr](#), [PMOVSSET](#), [PMSELR](#), [PMXEVCNTR](#), and [PMXEVTYPER](#).
- MRC reads of [PMCEID0](#) and [PMCEID1](#).
- MCR writes to [PMSWINC](#).
- MRRC or MCRR accesses to [PMCCNTR](#).

When trapped, reads and writes are UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMUSERENR

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMUSERENR();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = PMUSERENR();
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMUSERENR();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1110	0b000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
```

```
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    else
        PMUSERENR() = R(t);
    end;
elsif PSTATE.EL == EL2 then
    PMUSERENR() = R(t);
end;
```

D10.4.17 PMXEVNTR, Performance Monitors Selected Event Count Register

The PMXEVNTR characteristics are:

Purpose

Reads or writes the value of the selected event counter, [PMEVCNTR<n>](#). [PMSELR.SEL](#) determines which event counter is selected.

Configuration

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVNTR are UNDEFINED.

Attributes

PMXEVNTR is a 32-bit register.

Field descriptions



PMEVCNTR<n>, bits [31:0]

Value of the selected event counter, [PMEVCNTR<n>](#), where n is the value stored in [PMSELR.SEL](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMXEVNTR

If [PMSELR.SEL](#) is greater than or equal to the number of accessible event counters, then reads and writes of [PMEVCNTR](#) are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP
- Accesses to the register behave as if [PMSELR.SEL](#) has an UNKNOWN value less than the number of event counters accessible at the current Exception level and Security state.
- If [PMSELR.SEL](#) is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by [PMUSERENR.{ER,EN}](#) or [PMUSERENR_EL0.{ER,EN}](#).

At EL0 and EL1 [HDCR.HPMN](#) identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see [HDCR.HPMN](#).

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b010

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif UInt(PMSELR().SEL) >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR().[ER,EN] == '00' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        R(t) = PMEVCNTR(UInt(PMSELR().SEL));
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        R(t) = PMEVCNTR(UInt(PMSELR().SEL));
    end;
elsif PSTATE.EL == EL2 then
    R(t) = PMEVCNTR(UInt(PMSELR().SEL));
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b010

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif UInt(PMSELR().SEL) >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) >= GetNumEventCountersAccessible() then

```

```
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVCNTR(UInt(PMSELR().SEL)) = R(t);
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVCNTR(UInt(PMSELR().SEL)) = R(t);
    end;
elseif PSTATE.EL == EL2 then
    PMEVCNTR(UInt(PMSELR().SEL)) = R(t);
end;
```

D10.4.18 PMXEVTYPER, Performance Monitors Selected Event Type Register

The PMXEVTYPER characteristics are:

Purpose

When [PMSELR.SEL](#) selects an event counter, this accesses a [PMEVTYPER<n>](#) register. When [PMSELR.SEL](#) selects the cycle counter, this accesses [PMCCFILTR](#).

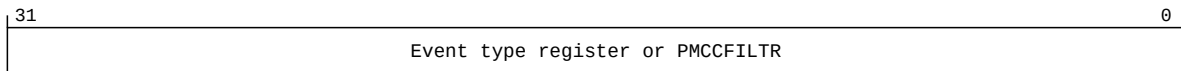
Configuration

This register is present only when FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMXEVTYPER are UNDEFINED.

Attributes

PMXEVTYPER is a 32-bit register.

Field descriptions



ETR, bits [31:0]

Event type register or [PMCCFILTR](#).

When [PMSELR.SEL](#) == 31, this register accesses [PMCCFILTR](#).

Otherwise, this register accesses [PMEVTYPER<n>](#) where n is the value in [PMSELR.SEL](#).

Accessing PMXEVTYPER

If [PMSELR.SEL](#) is not 31, and is greater than or equal to the number of accessible event counters, then reads and writes of [PMXEVTYPER](#) are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP
- Accesses to the register behave as if [PMSELR.SEL](#) has an UNKNOWN value less than the number of event counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if [PMSELR.SEL](#) is 31.
- If EL2 is implemented and enabled in the current Security state, and [PMSELR.SEL](#) is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by [PMUSERENR.EN](#) or [PMUSERENR_EL0.EN](#).

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1:

- If EL2 is using AArch32, [HDCR.HPMN](#) identifies the number of accessible event counters.
- If EL2 is using AArch64, [MDCR_EL2.HPMN](#) identifies the number of accessible event counters.

Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see [HDCR.HPMN](#) and [MDCR_EL2.HPMN](#).

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b001

```

if !IsFeatureImplemented(FEAT_PMUv3) then
    Undefined();
elsif UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >=
        ↪GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        if UInt(PMSELR().SEL) == 31 then
            R(t) = PMCCFILTR();
        else
            R(t) = PMEVTYPER(UInt(PMSELR().SEL));
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >=
        ↪GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        if UInt(PMSELR().SEL) == 31 then
            R(t) = PMCCFILTR();
        else
            R(t) = PMEVTYPER(UInt(PMSELR().SEL));
        end;
    end;
elsif PSTATE.EL == EL2 then
    if UInt(PMSELR().SEL) == 31 then
        R(t) = PMCCFILTR();
    else
        R(t) = PMEVTYPER(UInt(PMSELR().SEL));
    end;
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1001	0b1101	0b001

```

if !IsFeatureImplemented(FEAT_PMuV3) then
    Undefined();
elsif UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR().EN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >=
        ↪GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        if UInt(PMSELR().SEL) == 31 then
            PMCCFILTR() = R(t);
        else
            PMEVTYPER(UInt(PMSELR().SEL)) = R(t);
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HSTR().T9 == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && HDCR().TPM == '1' then
        AArch32_TakeHypTrapException(0x03);
    elsif EL2Enabled() && UInt(PMSELR().SEL) != 31 && UInt(PMSELR().SEL) >=
        ↪GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        if UInt(PMSELR().SEL) == 31 then
            PMCCFILTR() = R(t);
        else
            PMEVTYPER(UInt(PMSELR().SEL)) = R(t);
        end;
    end;
elsif PSTATE.EL == EL2 then
    if UInt(PMSELR().SEL) == 31 then
        PMCCFILTR() = R(t);
    else
        PMEVTYPER(UInt(PMSELR().SEL)) = R(t);
    end;
end;
end;

```

D10.5 Generic Timer registers

This section lists the Generic Timer registers in AArch32.

D10.5.1 CNTFRQ, Counter-timer Frequency register

The CNTFRQ characteristics are:

Purpose

This register is provided so that software can discover the [effective frequency](#) of the system counter. It must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

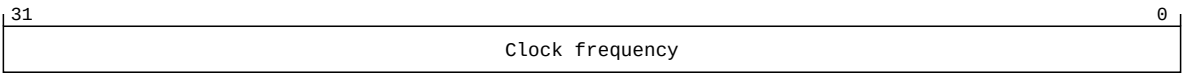
Configuration

There are no configuration notes.

Attributes

CNTFRQ is a 32-bit register.

Field descriptions



ClockFreq, bits [31:0]

Clock frequency. Indicates the [effective frequency](#) of the system counter, in Hz.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFRQ

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPCTEN == '0' && CNTKCTL().PL0VCTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    else
        R(t) = CNTFRQ();
    end;
elsif PSTATE.EL == EL1 then
    R(t) = CNTFRQ();
elsif PSTATE.EL == EL2 then
    R(t) = CNTFRQ();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0000	0b000

```
if IsHighestEL(PSTATE.EL) then
    CNTFRQ() = R(t);
else
    Undefined();
end;
```

D10.5.2 CNTHCTL, Counter-timer Hyp Control register

The CNTHCTL characteristics are:

Purpose

Controls the generation of an event stream from the physical counter, and access from Non-secure EL1 modes to the physical counter and the Non-secure EL1 physical timer.

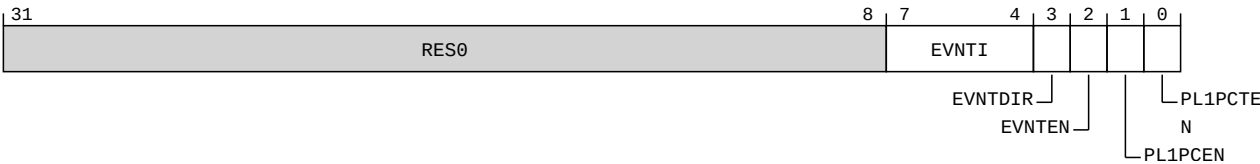
Configuration

There are no configuration notes.

Attributes

CNTHCTL is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

EVNTI, bits [7:4]

Selects which bit of CNTPCT, as seen from EL2, is the trigger for the event stream generated from that counter when that stream is enabled.

This field selects a trigger bit in the range 0 to 15 of CNTPCT.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the CNTPCT trigger bit, as seen from EL2 and defined by EVNTI, generates an event when the event stream is enabled.

EVNTDIR	Meaning
0b0	A 0 to 1 transition of the trigger bit triggers an event.
0b1	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from CNTPCT as seen from EL2.

EVNTEN	Meaning
0b0	Disables the event stream.
0b1	Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL1PCEN, bit [1]

Traps Non-secure EL0 and EL1 MRC or MCR accesses, reported using EC syndrome value 0x03, and MRRC or MCRR accesses, reported using EC syndrome value 0x04, to the physical timer registers to Hyp mode.

PL1PCEN	Meaning
0b0	Non-secure EL0 and EL1 accesses to the CNTP_CTL , CNTP_CVAL , and CNTP_TVAL are trapped to Hyp mode, unless trapped by CNTKCTL.PL0PTEN .
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL1PCTEN, bit [0]

Traps Non-secure EL0 and EL1 MRRC or MCRR accesses, reported using EC syndrome value 0x04, to the physical counter register to Hyp mode.

PL1PCTEN	Meaning
0b0	Non-secure EL0 and EL1 accesses to the CNTPCT are trapped to Hyp mode, unless it is trapped by CNTKCTL.PL0PCTEN .
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHCTL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    R(t) = CNTHCTL();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTHCTL() = R(t);
end;
```

D10.5.3 CNTHP_CTL, Counter-timer Hyp Physical Timer Control register

The CNTHP_CTL characteristics are:

Purpose

Control register for the Hyp mode physical timer.

Configuration

There are no configuration notes.

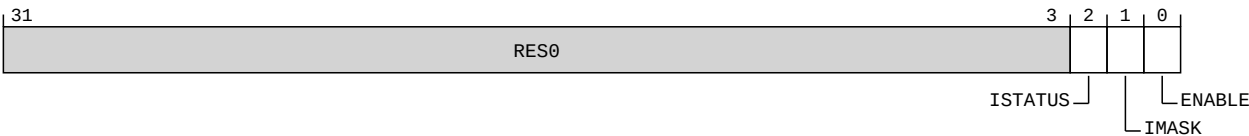
Attributes

CNTHP_CTL is a 32-bit register.

This register has the following instances:

- CNTHP_CTL.

Field descriptions



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTHP_TVAL](#) continues to count down.

Note
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing CNTHP_CTL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    R(t) = CNTHP_CTL();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b001

```

if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    Undefined();
elseif PSTATE.EL == EL2 then
    CNTHP_CTL() = R(t);
end;

```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    end;
elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
    AArch32_TakeHypTrapException(0x03);
else
    R(t) = CNTP_CTL();
end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CNTP_CTL();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = CNTP_CTL();
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    end;
elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then

```

```
        AArch32_TakeHypTrapException(0x03);
    else
        CNTP_CTL() = R(t);
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        CNTP_CTL() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    CNTP_CTL() = R(t);
end;
```

D10.5.4 CNTHP_CVAL, Counter-timer Hyp Physical CompareValue register

The CNTHP_CVAL characteristics are:

Purpose

Holds the compare value for the Hyp mode physical timer.

Configuration

There are no configuration notes.

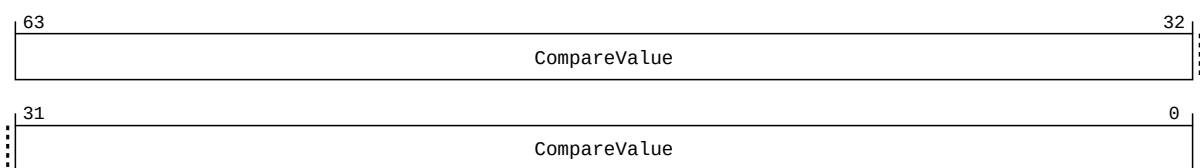
Attributes

CNTHP_CVAL is a 64-bit register.

This register has the following instances:

- CNTHP_CVAL.

Field descriptions



CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When [CNTHP_CTL.ENABLE](#) is 1, the timer condition is met when ([CNTPCT](#) - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- [CNTHP_CTL.ISTATUS](#) is set to 1.
- If [CNTHP_CTL.IMASK](#) is 0, an interrupt is generated.

When [CNTHP_CTL.ENABLE](#) is 0, the timer condition is not met, but [CNTPCT](#) continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_CVAL

Accesses to this register use the following encodings in the System register encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0110

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    R(t, t2) = CNTHP_CVAL();
end;

```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0110

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTHP_CVAL() = R(t, t2);
end;

```

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        R(t, t2) = CNTP_CVAL();
    end;
elsif PSTATE.EL == EL2 then
    R(t, t2) = CNTP_CVAL();
end;

```

MCCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elseif PSTATE.EL == EL2 then
    CNTP_CVAL() = R(t, t2);
end;
```


D10.5.5 CNTHP_TVAL, Counter-timer Hyp Physical Timer TimerValue register

The CNTHP_TVAL characteristics are:

Purpose

Holds the timer value for the Hyp mode physical timer.

Configuration

There are no configuration notes.

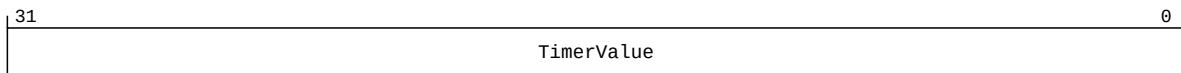
Attributes

CNTHP_TVAL is a 32-bit register.

This register has the following instances:

- CNTHP_TVAL.

Field descriptions



TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If [CNTHP_CTL.ENABLE](#) is 0, the value returned is UNKNOWN.
- If [CNTHP_CTL.ENABLE](#) is 1, the value returned is ([CNTHP_CVAL](#) - [CNTPCT](#)).

On a write of this register, [CNTHP_CVAL](#) is set to ([CNTPCT](#) + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When [CNTHP_CTL.ENABLE](#) is 1, the timer condition is met when ([CNTPCT](#) - [CNTHP_CVAL](#)) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- [CNTHP_CTL.ISTATUS](#) is set to 1.
- If [CNTHP_CTL.IMASK](#) is 0, an interrupt is generated.

When [CNTHP_CTL.ENABLE](#) is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHP_TVAL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    if CNTHP_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTHP_CVAL() - PhysicalCountInt())[31:0];
    end;
end;
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b100	0b1110	0b0010	0b000

```

if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTHP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
end;
end;

```

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elsif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then

```

```

        AArch32_TakeHypTrapException(0x03);
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elseif PSTATE.EL == EL2 then
    if CNTP_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
    end;
end;
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elseif PSTATE.EL == EL2 then
    CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
end;
end;

```

D10.5.6 CNTKCTL, Counter-timer Kernel Control register

The CNTKCTL characteristics are:

Purpose

Controls the generation of an event stream from the virtual counter, and access from EL0 modes to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

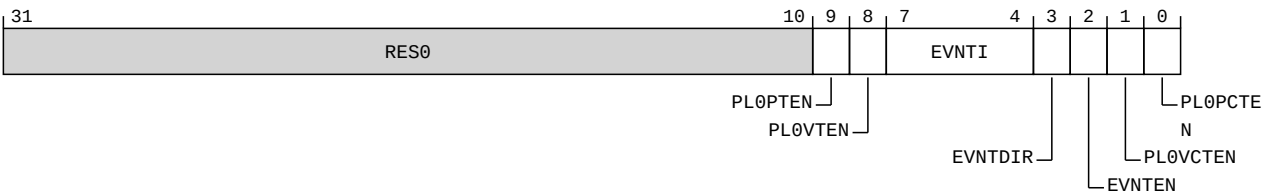
Configuration

There are no configuration notes.

Attributes

CNTKCTL is a 32-bit register.

Field descriptions



Bits [31:10]

Reserved, RES0.

PL0PTEN, bit [9]

Traps PL0 accesses to the physical timer registers to Undefined mode.

PL0PTEN	Meaning
0b0	PL0 accesses to the CNTP_CTL , CNTP_CVAL , and CNTP_TVAL registers are trapped to Undefined mode.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL0VTEN, bit [8]

Traps PL0 accesses to the virtual timer registers to Undefined mode.

PL0VTEN	Meaning
0b0	PL0 accesses to the CNTV_CTL , CNTV_CVAL , and CNTV_TVAL registers are trapped to Undefined mode.

PL0VTEN	Meaning
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTI, bits [7:4]

Selects which bit of [CNTVCT](#), as seen from EL1, is the trigger for the event stream generated from that counter when that stream is enabled.

This field selects a trigger bit in the range 0 to 15 of [CNTVCT](#).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the [CNTVCT](#) trigger bit, as seen from EL1 and defined by EVNTI, generates an event when the event stream is enabled.

EVNTDIR	Meaning
0b0	A 0 to 1 transition of the trigger bit triggers an event.
0b1	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from [CNTVCT](#) as seen from EL1.

EVNTEN	Meaning
0b0	Disables the event stream.
0b1	Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL0VCTEN, bit [1]

Traps PL0 accesses to the frequency register and virtual counter register to Undefined mode.

PL0VCTEN	Meaning
0b0	PL0 accesses to the CNTVCT are trapped to Undefined mode. PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0PCTEN is also 0.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PL0PCTEN, bit [0]

Traps PL0 accesses to the frequency register and physical counter register to Undefined mode.

PL0PCTEN	Meaning
0b0	PL0 accesses to the CNTPCT are trapped to Undefined mode. PL0 accesses to the CNTFRQ register are trapped to Undefined mode, if CNTKCTL.PL0VCTEN is also 0.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTKCTL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    R(t) = CNTKCTL();
elsif PSTATE.EL == EL2 then
    R(t) = CNTKCTL();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    CNTKCTL() = R(t);
elsif PSTATE.EL == EL2 then
    CNTKCTL() = R(t);
end;
```

D10.5.7 CNTP_CTL, Counter-timer Physical Timer Control register

The CNTP_CTL characteristics are:

Purpose

Control register for the EL1 physical timer.

Configuration

There are no configuration notes.

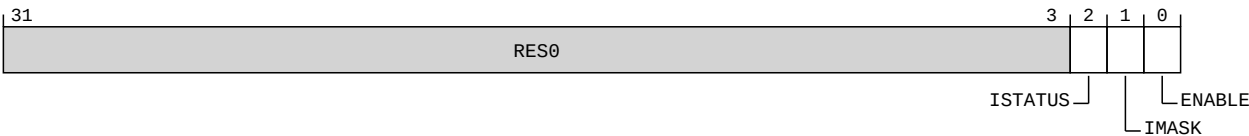
Attributes

CNTP_CTL is a 32-bit register.

This register has the following instances:

- CNTP_CTL.

Field descriptions



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTP_TVAL](#) continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing CNTP_CTL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CNTP_CTL();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
```

```
        AArch32_TakeHypTrapException(0x03);
    else
        R(t) = CNTP_CTL();
    end;
elseif PSTATE.EL == EL2 then
    R(t) = CNTP_CTL();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        CNTP_CTL() = R(t);
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        CNTP_CTL() = R(t);
    end;
elseif PSTATE.EL == EL2 then
    CNTP_CTL() = R(t);
end;
```

D10.5.8 CNTP_CVAL, Counter-timer Physical Timer CompareValue register

The CNTP_CVAL characteristics are:

Purpose

Holds the compare value for the EL1 physical timer.

Configuration

There are no configuration notes.

Attributes

CNTP_CVAL is a 64-bit register.

This register has the following instances:

- CNTP_CVAL.

Field descriptions



CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When [CNTP_CTL.ENABLE](#) is 1, the timer condition is met when ([CNTPCT](#) - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- [CNTP_CTL.ISTATUS](#) is set to 1.
- If [CNTP_CTL.IMASK](#) is 0, an interrupt is generated.

When [CNTP_CTL.ENABLE](#) is 0, the timer condition is not met, but [CNTPCT](#) continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL

Accesses to this register use the following encodings in the System register encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        R(t, t2) = CNTP_CVAL();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        R(t, t2) = CNTP_CVAL();
    end;
elseif PSTATE.EL == EL2 then
    R(t, t2) = CNTP_CVAL();
end;

```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0010

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        CNTP_CVAL() = R(t, t2);
    end;
elseif PSTATE.EL == EL2 then
    CNTP_CVAL() = R(t, t2);
end;

```

D10.5.9 CNTP_TVAL, Counter-timer Physical Timer TimerValue register

The CNTP_TVAL characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configuration

There are no configuration notes.

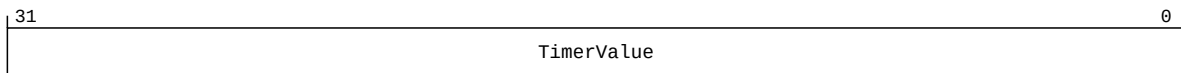
Attributes

CNTP_TVAL is a 32-bit register.

This register has the following instances:

- CNTP_TVAL.

Field descriptions



TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If [CNTP_CTL.ENABLE](#) is 0, the value returned is UNKNOWN.
- If [CNTP_CTL.ENABLE](#) is 1, the value returned is ([CNTP_CVAL](#) - [CNTPCT](#)).

On a write of this register, [CNTP_CVAL](#) is set to ([CNTPCT](#) + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When [CNTP_CTL.ENABLE](#) is 1, the timer condition is met when ([CNTPCT](#) - [CNTP_CVAL](#)) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- [CNTP_CTL.ISTATUS](#) is set to 1.
- If [CNTP_CTL.IMASK](#) is 0, an interrupt is generated.

When [CNTP_CTL.ENABLE](#) is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_TVAL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        if CNTP_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
        end;
    end;
elseif PSTATE.EL == EL2 then
    if CNTP_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTP_CVAL() - PhysicalCountInt())[31:0];
    end;
end;
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0010	0b000

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCEN == '0' then
        AArch32_TakeHypTrapException(0x03);
    else

```

```
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();  
    end;  
    elsif PSTATE.EL == EL2 then  
        CNTP_CVAL() = SignExtend{64}(R(t)) + PhysicalCountInt();  
    end;
```

D10.5.10 CNTPCT, Counter-timer Physical Count register

The CNTPCT characteristics are:

Purpose

Holds the 64-bit physical count value.

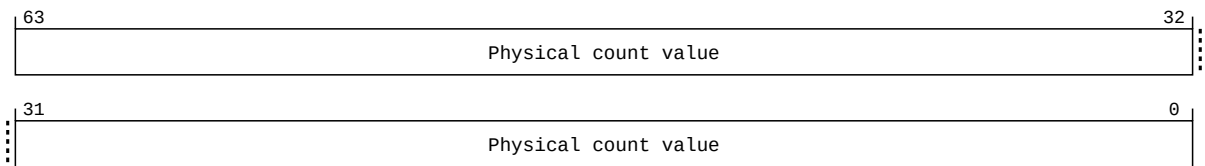
Configuration

There are no configuration notes.

Attributes

CNTPCT is a 64-bit register.

Field descriptions



PhysicalCount, bits [63:0]

Physical count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCT

Accesses to this register use the following encodings in the System register encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0000

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOPCTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    elseif EL2Enabled() && CNTHCTL().PL1PCTEN == '0' then
        AArch32_TakeHypTrapException(0x04);
    else
        R(t, t2) = PhysicalCountInt();
    end;
elseif PSTATE.EL == EL1 then
    if EL2Enabled() && CNTHCTL().PL1PCTEN == '0' then
        AArch32_TakeHypTrapException(0x04);
```



```
        else
            R(t, t2) = PhysicalCountInt();
        end;
    elsif PSTATE.EL == EL2 then
        R(t, t2) = PhysicalCountInt();
    end;
```

D10.5.11 CNTV_CTL, Counter-timer Virtual Timer Control register

The CNTV_CTL characteristics are:

Purpose

Control register for the virtual timer.

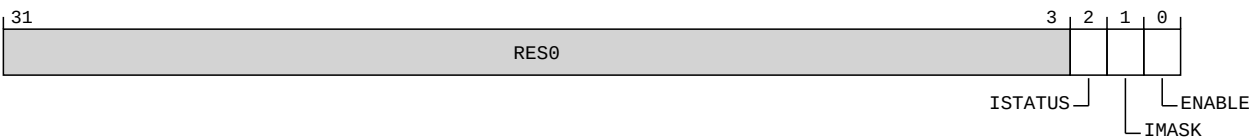
Configuration

There are no configuration notes.

Attributes

CNTV_CTL is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

Note
Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing CNTV_CTL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOVTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    else
        R(t) = CNTV_CTL();
    end;
elsif PSTATE.EL == EL1 then
    R(t) = CNTV_CTL();
elsif PSTATE.EL == EL2 then
    R(t) = CNTV_CTL();
end;
```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOVTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    else
        CNTV_CTL() = R(t);
    end;
elsif PSTATE.EL == EL1 then
    CNTV_CTL() = R(t);
elsif PSTATE.EL == EL2 then
    CNTV_CTL() = R(t);
end;
```

D10.5.12 CNTV_CVAL, Counter-timer Virtual Timer CompareValue register

The CNTV_CVAL characteristics are:

Purpose

Holds the compare value for the virtual timer.

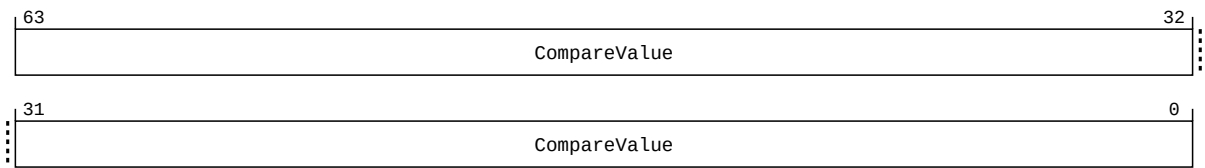
Configuration

There are no configuration notes.

Attributes

CNTV_CVAL is a 64-bit register.

Field descriptions



CompareValue, bits [63:0]

Holds the EL1 virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CVAL

Accesses to this register use the following encodings in the System register encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOVTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    else
        R(t, t2) = CNTV_CVAL();
    end;
elseif PSTATE.EL == EL1 then
    R(t, t2) = CNTV_CVAL();
elseif PSTATE.EL == EL2 then
    R(t, t2) = CNTV_CVAL();
end;
```

MCCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0011

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOVTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    else
        CNTV_CVAL() = R(t, t2);
    end;
elseif PSTATE.EL == EL1 then
    CNTV_CVAL() = R(t, t2);
elseif PSTATE.EL == EL2 then
    CNTV_CVAL() = R(t, t2);
end;
```

D10.5.13 CNTV_TVAL, Counter-timer Virtual Timer TimerValue register

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

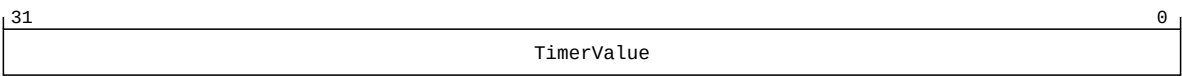
Configuration

There are no configuration notes.

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions



TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If `CNTV_CTL.ENABLE` is 0, the value returned is UNKNOWN.
- If `CNTV_CTL.ENABLE` is 1, the value returned is `(CNTV_CVAL - CNTVCT)`.

On a write of this register, `CNTV_CVAL` is set to `(CNTVCT + TimerValue)`, where `TimerValue` is treated as a signed 32-bit integer.

When `CNTP_CTL.ENABLE` is 1, the timer condition is met when `(CNTVCT - CNTP_CVAL)` is greater than or equal to zero. This means that `TimerValue` acts like a 32-bit downcounter timer. When the timer condition is met:

- `CNTV_CTL.ISTATUS` is set to 1.
- If `CNTV_CTL.IMASK` is 0, an interrupt is generated.

When `CNTV_CTL.ENABLE` is 0, the `TimerValue` cannot be read but continues to decrement. When the timer is enabled, the `TimerValue` represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_TVAL

Accesses to this register use the following encodings in the System register encoding space:

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOVTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    else
        if CNTV_CTL().ENABLE == '0' then
            R(t) = ARBITRARY:bits(32);
        else
            R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
        end;
    end;
elseif PSTATE.EL == EL1 then
    if CNTV_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
    end;
elseif PSTATE.EL == EL2 then
    if CNTV_CTL().ENABLE == '0' then
        R(t) = ARBITRARY:bits(32);
    else
        R(t) = (CNTV_CVAL() - (PhysicalCountInt() - CNTVOFF()))[31:0];
    end;
end;

```

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

coproc	opc1	CRn	CRm	opc2
0b1111	0b000	0b1110	0b0011	0b000

```

if PSTATE.EL == EL0 then
    if CNTKCTL().PLOVTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    else
        CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
    end;
elseif PSTATE.EL == EL1 then
    CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
elseif PSTATE.EL == EL2 then
    CNTV_CVAL() = (SignExtend{64}(R(t)) + PhysicalCountInt()) - CNTVOFF();
end;

```


D10.5.14 CNTVCT, Counter-timer Virtual Count register

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value minus the virtual offset visible in CNTVOFF.

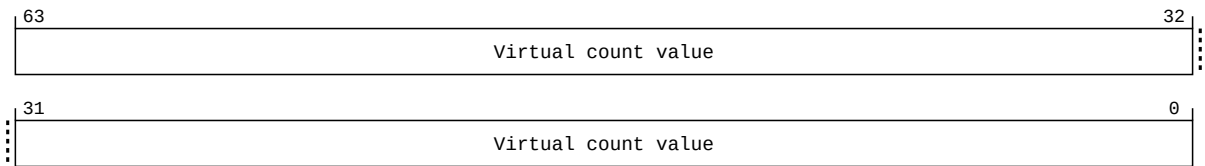
Configuration

There are no configuration notes.

Attributes

CNTVCT is a 64-bit register.

Field descriptions



VirtualCount, bits [63:0]

Virtual count value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCT

Accesses to this register use the following encodings in the System register encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0001

```
if PSTATE.EL == EL0 then
    if CNTKCTL().PLOVCTEN == '0' then
        if EL2Enabled() && HCR().TGE == '1' then
            AArch32_TakeHypTrapException(0x00);
        else
            Undefined();
        end;
    else
        R(t, t2) = PhysicalCountInt() - CNTVOFF();
    end;
elsif PSTATE.EL == EL1 then
    R(t, t2) = PhysicalCountInt() - CNTVOFF();
elsif PSTATE.EL == EL2 then
    R(t, t2) = PhysicalCountInt() - CNTVOFF();
end;
```

D10.5.15 CNTVOFF, Counter-timer Virtual Offset register

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset between the physical count value visible in CNTPCT and the virtual count value visible in CNTVCT.

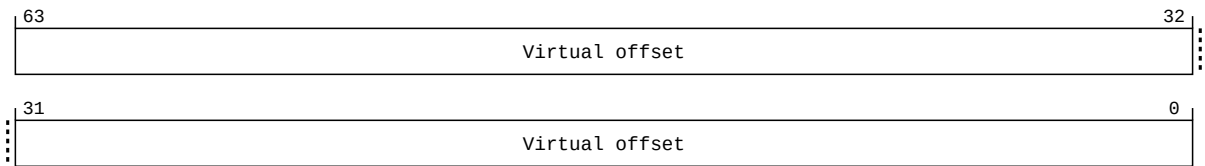
Configuration

There are no configuration notes.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions



VOffset, bits [63:0]

Virtual offset.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF

Accesses to this register use the following encodings in the System register encoding space:

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0100

```
if PSTATE.EL == EL0 then
    Undefined();
elseif PSTATE.EL == EL1 then
    Undefined();
elseif PSTATE.EL == EL2 then
    R(t, t2) = CNTVOFF();
end;
```

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

coproc	CRm	opc1
0b1111	0b1110	0b0100

```
if PSTATE.EL == EL0 then
    Undefined();
elsif PSTATE.EL == EL1 then
    Undefined();
elsif PSTATE.EL == EL2 then
    CNTVOFF() = R(t, t2);
end;
```

Part E

External Debug

Chapter E1

About External Debug

This chapter gives an overview of A-profile external debug and specifies the required debug authentication. It contains the following sections:

- [Introduction to external debug](#)
- [External debug](#)
- [Required debug authentication](#)

Note

For information about self-hosted debug, see [AArch32 Self-hosted Debug](#).

E1.1 Introduction to external debug

The architecture supports both:

Self-hosted debug

The PE itself hosts a debugger. That is, developers developing software to run on the PE use debugger software running on the same PE.

External debug

The debugger is external to the PE. The debugging might be either on-chip, for example in a second PE, or off-chip, for example a JTAG debugger that accesses the chip through a Debug Access Port.

External debug is particularly useful for:

- Hardware bring-up. That is, debugging during development when a system is first powered up and not all of the software functionality is available.
- PEs that are deeply embedded inside systems.

To support external debug, the Arm architecture defines required features that are called *external debug features*.

Note

An external debugger has a potentially high level of control over and visibility into the PE. The system sets this level using debug authentication. See [Required debug authentication](#). If the debug authentication level is set too low, agents may be able to bypass elements of the security and privilege models. This includes both off-chip agents and on-chip agents such as unprivileged or Non-secure software.

E1.1.1 Definition and constraints of a debugger in the context of external debug

When the description of external debug in this Part of the manual describes a *debugger* as controlling external debug this debugger might be a second on-chip PE or an off-chip device such as a JTAG debugger using a Debug Access Port (DAP).

If a Debug Access Port is implemented:

- When debug is prohibited at the Debug Access Port, the port must not generate accesses to the external debug interface of the PE.

Accesses to the PE are controlled by the external authentication interface functions, [ExternalInvasiveDebugEnabled\(\)](#) and [ExternalNoninvasiveDebugEnabled\(\)](#).

The Debug Access Port is not required to use the same authentication interface as the PE.

Arm recommends that when [ExternalInvasiveDebugEnabled\(\)](#) == FALSE at the PE, all debug is prohibited at the DAP.

See also [Required debug authentication](#).

E1.2 External debug

Debug events allow an external debugger to halt the PE. The provides the following debug events:

- [Halting Step debug events:](#)
 - The debugger can use this resource to make the PE step through code one line at a time.
- [Halt Instruction debug event:](#)
 - This might occur when software executes the Halting breakpoint instruction, `HLT`.
- [Exception Catch debug event:](#)
 - This can be programmed to occur on all entries to a given Exception level.
- [External Debug Request debug event:](#)
 - An embedded cross-trigger can signal this debug event.
- [OS Unlock Catch debug event:](#)
 - This might occur when the state of the OS Lock changes from locked to unlocked.
- [Reset Catch debug events:](#)
 - This might occur when the PE exits reset state.
- [Software Access debug event:](#)
 - This can be programmed to occur when software tries to access the Breakpoint Value registers, the Breakpoint Control registers, the Watchpoint value registers, or the Watchpoint Control registers. It caused a trap to Debug state.

Breakpoints and watchpoints can also halt the PE.

When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register*, ITR, passes instructions to the PE to execute in Debug state:
 - The ITR contains a single register, EDITR, and associated flow-control flags.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger:
 - The DCC includes the data transfer registers, DTRRX and DTRTX, and associated flow-control flags.
 - Although the DCC is an essential part of Debug state operation, it can also be used in Non-debug state.
- The PE cannot service any interrupts in Debug state.

[Debug State](#) describes Debug state in more detail.

E1.3 Required debug authentication

A PE implements the debug authentication interface defined in this section. The debug authentication interface controls:

- Whether halting the PE and entering Debug state is allowed or prohibited.
- Whether some aspects of non-invasive debug are allowed or prohibited.
- Some legacy aspects of the AArch32 self-hosted debug model.

The pseudocode functions shown in [Table E1-1](#), and the conditions that follow that table, define the architectural requirements for debug authentication.

Table E1-1 Debug authentication functions

Pseudocode function	Description
ExternalNoninvasiveDebugEnabled()	Returns TRUE if non-invasive debug is enabled.
ExternalInvasiveDebugEnabled()	Returns TRUE if invasive debug is enabled.

The following conditions always apply:

- If [ExternalInvasiveDebugEnabled\(\)](#) is TRUE then [ExternalNoninvasiveDebugEnabled\(\)](#) is TRUE.

Arm recommends the use of the interface described in Recommended authentication interface to provide this debug authentication. The pseudocode functions in [R-profile Architecture Pseudocode](#), which are linked to by the entries in the [Pseudocode function](#) column of [Table E1-1](#), assume that this interface is implemented.

Chapter E2

Debug State

This chapter describes Debug state. It contains the following sections:

- [About Debug state](#)
- [Halting the PE on debug events](#)
- [Entering Debug state](#)
- [Behavior in Debug state](#)
- [Exiting Debug state](#)

E2.1 About Debug state

In external debug, debug events allow an external debugger to halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- It stops executing instructions from the location indicated by the Program Counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register*, ITR, passes instructions to the PE to execute in Debug state.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger.

The PE cannot service any interrupts in Debug state.

E2.2 Halting the PE on debug events

For details of debug events, see [Introduction to Halting debug events](#) and [Breakpoint and Watchpoint debug events](#).

On a debug event, the PE must do one of the following:

- Enter Debug state.
- Pend the debug event.
- Generate a debug exception.
- Ignore the debug event.

This behavior depends on both:

- Whether halting is allowed by the current state of the debug authentication interface. See [Halting allowed and halting prohibited](#).
- The type of debug event and the programming of the debug control registers.
 - See [Halting debug events](#) for all Halting debug events.
 - See [Breakpoint and Watchpoint debug events](#) for Breakpoint and Watchpoint debug events.

See also [Other debug exceptions](#).

This means that behavior can be CONSTRAINED UNPREDICTABLE if the conditions change. See [Synchronization and Halting debug events](#).

[Table E3-1](#) summarizes the possible outcomes of each type of debug event.

E2.2.1 Halting allowed and halting prohibited

Halting can be either allowed or prohibited:

- Halting is always prohibited in Debug state.
- Halting is always prohibited when `DoubleLockStatus() == TRUE`.
 - This means that `FEAT_DoubleLock` is implemented and OS Double lock is locked.
- Otherwise, halting is allowed.

For more information, see:

- [Pseudocode description of Halting on debug events](#).
- [Required debug authentication](#).
- Recommended External Debug Interface.

E2.2.2 Halting debug events

The Halting debug events are described in [Halting Debug Events](#).

When a Halting debug event is generated, it causes entry to Debug state if all of the following are true:

- Halting is allowed. See [Halting allowed and halting prohibited](#).
- The Halting debug event is one of:
 - A Halt Instruction debug event and `EDSCR.HDE == 1`.
 - A Software Access debug event and `OSLSR_EL1.OSLK == 0`, meaning that the OS Lock is unlocked.
 - Neither a Halt Instruction debug event nor a Software Access debug event.

Note

- A Halt Instruction debug event is the only Halting debug event that relies on [EDSCR.HDE == 1](#).
 - Halting on Breakpoint and Watchpoint debug events is also controlled by [EDSCR.HDE](#). See [Breakpoint and Watchpoint debug events](#).
 - [EDSCR.HDE](#) can be written by software when the OS Lock is locked. Privileged code can use [HDCR.TDOSA](#) to trap writes to OS Lock registers.
-

If a Halting debug event does not generate entry to Debug state because the conditions listed in this section do not hold, then:

- If the Halting debug event is a Halt Instruction debug event, the instruction that generated the Halting debug event is treated as UNDEFINED.
- If the Halting debug event is a Software Access debug event, it is ignored.
- If the Halting debug event is an Exception Catch debug event, it is ignored except when FEAT_Debugv8p8 is implemented, in which case the event might be pended.

In all other cases the Halting debug event is pended, see [Pending Halting debug events](#).

[Summary of actions from debug events](#) summarizes the possible outcome for each type of Debug event.

Note

Halting debug events never generate debug exceptions.

E2.2.3 Breakpoint and Watchpoint debug events

A breakpoint or watchpoint generates an entry to Debug state if all of the following conditions hold:

- Halting debug is enabled, that is [EDSCR.HDE == 1](#).
- Halting is allowed. See [Halting allowed and halting prohibited](#).
- The OS Lock is unlocked, that is [DBGOSLSR.OSLK == 0](#).

The Address Mismatch breakpoint type is reserved when all of these conditions are met.

[DBGDSCRExt.MDBGen](#) is ignored when determining whether to enter Debug state. A breakpoint or watchpoint that generates entry to Debug state is a Breakpoint or Watchpoint debug event and does not generate a debug exception.

A breakpoint or watchpoint that does not generate an entry to Debug state either:

- Generates a Breakpoint or Watchpoint exception.
- Is ignored.

Note

[EDSCR.HDE](#) is ignored when determining whether to generate a debug exception. The debug exception is suppressed only if the PE enters Debug state. This means that the use of Halting debug mode in one Security state does not affect the Exception model in another Security state.

See [AArch32 Self-hosted Debug](#), and the Note in [Other debug exceptions](#).

E2.2.4 Other debug exceptions

The following events never generate entry to Debug state:

- Breakpoint Instruction exceptions.
- Software Step exceptions.

- Vector Catch exceptions.

The behavior of these events is unchanged when Halting debug mode is enabled, that is when `EDSCR.HDE == 1`. This means that these events can do one of the following:

- They can generate a debug exception.
- They can be ignored.

For additional information, see [AArch32 Self-hosted Debug](#).

E2.2.5 Debug state entry and debug event prioritization

The following are synchronous debug events:

- Breakpoint debug event.
- Watchpoint debug event.
- Halting Step debug event.
- Halt Instruction debug event.
- Exception Catch debug event.
- Software Access debug event.
- Reset Catch debug event.

Each of these synchronous debug events are treated as a synchronous exception generated by an instruction, or by the taking of an exception or reset. That is, if halting is allowed, the synchronous debug event must be taken before any subsequent instructions are executed. Reset Catch debug events must be taken before the PE executes the instruction at the reset vector.

Note

- Reset Catch and Exception Catch debug events might be generated asynchronously, because they can result from an asynchronous exception. However, if halting is allowed after the reset or asynchronous exception has been processed, the Reset Catch or Exception Catch debug event is taken synchronously.
 - The Halting Step debug event is generated by the instruction after the stepped instruction. Therefore, if the stepped instruction generates any other synchronous exceptions or debug events these are taken first.
-

If halting is prohibited then Halting Step debug events and Reset Catch debug events might be pended and taken asynchronously. OS Unlock Catch debug events are always pended and taken asynchronously. See [Pending Halting debug events](#).

The architecture does not define when asynchronous debug events are taken, and therefore the prioritization of asynchronous debug events is IMPLEMENTATION DEFINED. See [Synchronization and Halting debug events](#).

The following list shows how the synchronous debug events are prioritized, with 1 being the highest priority.

Note

The priority numbering is the same as the numbering for AArch64 synchronous exception priorities listed in Synchronous exception types, and in particular Prioritization of asynchronous exceptions. This numbering correlates with the equivalent AArch32 list in [Exception prioritization for exceptions taken to AArch32 state](#).

The priority for synchronous debug events is as follows:

- 1 Reset Catch debug event. See [Reset Catch debug events](#).

This debug event has a higher priority than the synchronous exceptions listed in Synchronous exception types.
- 2 Exception Catch debug event if it has a priority of 2. See [Exception Catch debug event](#).

This debug event can be assigned one of two priorities, 2 or 5. When it has a priority of 2, it has a higher priority than the synchronous exceptions listed in the Exception model.

3	Halting Step debug event. See Halting Step debug events . This debug event has a higher priority than the synchronous exceptions listed in the Exception model.
4	This event is not a debug event.
5	Exception Catch debug event if it has a priority of 5. See Exception Catch debug event . This debug event can be assigned one of two priorities, 2 or 5.
6-9	These events are not debug events.
10	Breakpoint debug events. See Breakpoint and Watchpoint debug events .
11-13	These events are not debug events.
14	Halt Instruction debug event. See Halt Instruction debug event .
15-40	These events are not debug events.
41	Software Access debug event. See Software Access debug event .
42-46	These events are not debug events.
47	Watchpoint debug events. See Breakpoint and Watchpoint debug events .
48-49	These events are not debug events.

For Reset Catch debug events and Halting Step debug events, the priorities listed in this section apply only when halting is allowed at the time the event is generated. This means that the event is taken synchronously and not pended.

For more information on the prioritization of exceptions, see:

- Synchronous exception types.
- Prioritization of asynchronous exceptions.
- [Exception prioritization for exceptions taken to AArch32 state](#). This section covers synchronous and asynchronous exceptions.

E2.2.5.1 Breakpoint debug events and Vector Catch exception

An Address Matching Vector Catch exception has the same priority as a Breakpoint debug event. See Prioritization of Synchronous exceptions taken to AArch64 state.

The prioritization of these events is unchanged even if the breakpoint generates entry to Debug state instead of a Breakpoint exception. This means that if a single instruction generates both an Address Matching Vector Catch exception and a Breakpoint debug event, there is a CONSTRAINED UNPREDICTABLE choice of:

- The PE entering Debug state due to the Breakpoint debug event.
- A Vector Catch exception.

This applies only if all of the following are true:

- Halting debug is enabled.
- Halting is allowed.
- The OS Lock is unlocked.

An Exception Trapping Vector Catch exception must be generated immediately following the exception that generated it. This means that it does not appear in the priority table.

E2.2.6 Imprecise entry to Debug state

Debug state entry is normally *precise*. This means that the PE cannot enter Debug state if it can neither complete nor abandon all currently executing instructions and leave the PE in a precise state. See Definition of a precise exception and imprecise exception.

A debugger can write a value of 1 to [EDRCCR.CBRRQ](#) to allow imprecise entry to Debug state. An External Debug Request debug event must be pending before writing 1 to this bit. If 1 is written when an External Debug Request debug event is not pending, and the PE does not ignore the write, then the resulting PE behavior is UNPREDICTABLE. Support for allowing imprecise entry to Debug state using [EDRCCR.CBRRQ](#) is OPTIONAL, and it is IMPLEMENTATION DEFINED when this mechanism is effective at forcing entry to Debug state.

Arm recommends that the PE ignores writes to [EDRCCR.CBRRQ](#) when an External Debug Request debug event is not pending.

If forcing this PE to enter Debug state imprecisely could have an adverse effect on the wider system, then Arm recommends that a write to [EDRCCR.CBRRQ](#) is ignored if the highest level of debug authentication is not enabled at the system level.

[Example E2-1](#) shows how entry to Debug state can be forced.

Example E2-1 Forcing entry to Debug state

The debugger pends an External Debug Request debug event through the CTI to halt a program that has stopped responding. However, the memory system is not responding and a memory access instruction cannot complete. This means that Debug state cannot be entered precisely. The debugger writes a value of 1 to [EDRCCR.CBRRQ](#). The PE cancels all outstanding memory accesses and enters Debug state. As some instructions might not have completed correctly, entry to Debug state is imprecise.

When Debug state is entered imprecisely, all memory access instructions executed through the ITR have CONSTRAINED UNPREDICTABLE behavior. The value of all registers is UNKNOWN, but might be useful for diagnostic purposes.

E2.2.7 Summary of actions from debug events

[Table E2-1](#) shows the Software and Halting debug events. In [Table E2-1](#), the columns have the following meaning:

DLK	This indicates whether FEAT_DoubleLock is implemented and locked, DoubleLockStatus() == TRUE.
OSLK	This is the value of DBGOSLSR.OSLK . It indicates whether the OS Lock is locked.
HDE	This is the value of EDSCR.HDE . It indicates whether Halting debug is enabled.

The letter X in [Table E2-1](#) indicates that the value can be either 0 or 1.

Table E2-1 Debug authentication for external debug

Debug event type	Authentication	DLK	OSLK	HDE	Behavior
Other software	X	X	X	X	Handled by the Exception model
Breakpoint or Watchpoint debug event	X	TRUE	X	X	Handled by the Exception model (ignored)
	X	FALSE	1	X	Handled by the Exception model (ignored)
	FALSE	FALSE	0	X	Handled by the Exception model
	TRUE	FALSE	0	0	Handled by the Exception model

Debug event type	Authentication	DLK	OSLK	HDE	Behavior
Halt Instruction debug event	TRUE	FALSE	0	1	Entry to Debug state
	FALSE	X	X	X	UNDEFINED
	TRUE	TRUE	X	X	UNDEFINED
	TRUE	FALSE	X	0	UNDEFINED
Exception Catch debug event	TRUE	FALSE	X	1	Entry to Debug state
	FALSE	X	X	X	Might be pended ^a
	TRUE	TRUE	X	X	Ignored
Software Access debug event	TRUE	FALSE	X	X	Entry to Debug state
	FALSE	X	X	X	Ignored
	TRUE	TRUE	X	X	Ignored
	TRUE	FALSE	1	X	Ignored
Other Halting	TRUE	FALSE	0	X	Entry to Debug state
	FALSE	X	X	X	Debug event is pended
	TRUE	TRUE	X	X	Debug event is pended
	TRUE	FALSE	X	X	Entry to Debug state

a. See [Exception Catch debug event](#).

E2.2.8 Pseudocode description of Halting on debug events

The `Halted()`, `Restarting()`, `HaltingAllowed()`, and `HaltOnBreakpointOrWatchpoint()` functions are described in the pseudocode.

E2.3 Entering Debug state

On entry to Debug state, the preferred restart address and PSTATE are saved in [DLR](#) and [DSPSR](#). The PE remains in the mode and Security state from which it entered Debug state.

If [EDRCR.CBRRQ](#) has a value of 0, entry to Debug state is precise. If [EDRCR.CBRRQ](#) has a value of 1, then imprecise entry to Debug state is permitted.

If a Watchpoint debug event causes an entry to Debug state, the PE records the following:

- The address of the access that generated the Watchpoint debug event is recorded in [EDWAR](#).

For more information, see:

- [Exception syndrome information, fault address information, and preferred return address](#) for a debug event taken from AArch32 state.

Other than the effect on PSTATE and [EDSCR](#), entry to Debug state is not a [Context Synchronization event](#). The effects of entry to Debug state on PSTATE and [EDSCR](#) are synchronized.

E2.3.1 Entering Debug state from AArch32 state

When entering Debug state from AArch32 state, the PE remains in AArch32 state. In AArch32 Debug state the PE executes T32 instructions, regardless of the value of PSTATE.T before entering Debug state.

To allow the debugger to determine the state of the PE, the current Execution state for all four Exception levels can be read from [EDSCR.RW](#), and the current Exception level can be read from [EDSCR.EL](#).

The current endianness state, PSTATE.E, is unchanged on entry to Debug state.

Note

- If EL1 is using AArch32 state, the current endianness state can differ from that indicated by [SCTLR.EE](#).
 - If EL2 is using AArch32 state, the current endianness state can differ from that indicated by [HSCTLR.EE](#).
 - On entry to Debug state from AArch32 state, PSTATE.SS is copied to DSPSR.SS, even though the PE remains in AArch32 state.
-

See also [Effect of entering Debug state on PSTATE](#).

E2.3.2 Effect of Debug state entry on DLR and DSPSR

[DLR](#) is set to the preferred restart address for the debug event, which depends on the event type. The value of PSTATE is saved in [DSPSR](#).

For entry to Debug state from AArch32 state, the values saved in [DSPSR.IT](#) are always correct for the preferred restart address.

For synchronous Halting debug events, the preferred restart address is the address of the instruction that generated the debug event.

For asynchronous Halting debug events, including pending Halting debug events taken asynchronously, the preferred restart address is the address of the first instruction that must be executed on exit from Debug state.

This means that:

- For Breakpoint and Watchpoint debug events, the preferred restart address is the same as the preferred return address for a debug exception, as described in AArch64 Self-hosted Debug and [AArch32 Self-hosted Debug](#).
- For Halt Instruction debug events, [DLR](#) is set to the address of the `HLT` instruction and [DSPSR.IT](#) is correct for the `HLT` instruction.
- For Software Access debug events, [DLR](#) is set to the address of the accessing instruction and [DSPSR.IT](#) is correct for this instruction.

- For Halting Step debug events taken synchronously, [DLR](#) and [DSPSR](#) are set as the ELR and SPSR would be set for a Software Step exception. This is usually the address of, and PSTATE for, the instruction after the one that was stepped.
- For Exception Catch debug events, [DLR](#) is set to the address of the first instruction that must be executed on exit from Debug state, and [DSPSR](#) is correct for this instruction. This is the exception vector or reset vector address.
 - If the debug event is generated on taking an exception to a trapped Exception level, the [DLR](#) is set to the address of the exception vector the PE would have started fetching from. This is UNKNOWN if the VBAR for the Exception level has never been initialized. The [DSPSR](#) records the value of PSTATE after taking the exception. The Exception Catch occurs after the SPSR and the Link register are set, and the debugger can use these registers to determine where in the application program the exception occurred.

Note

Depending on the target Exception level and Execution state for the exception, the Link register is one of [ELR_hyp](#), or LR (R14).

- If the debug event is generated on an exception return to a trapped Exception level, the [DLR](#) is set to the target address of the exception return and the [DSPSR](#) records the value of PSTATE after the exception return.
- Reset Catch debug events taken synchronously behave like Exception Catch debug events.
- For Reset Catch debug events and Exception Catch debug events generated on reset to a trapped Exception level, the [DLR](#) is set to the reset address and the [DSPSR](#) records the reset value of PSTATE.
- For pending Halting debug events and External Debug Request debug events, [DLR](#) is set to the address of the first instruction that must be executed on exit from Debug state and [DSPSR](#). It is correct for this instruction. See [Pending Halting debug events](#).

Normally [DLR](#) is aligned according to the instruction set state indicated in [DSPSR](#). However, a debug event might be taken at a point where the PC is not aligned.

E2.3.3 Effect of Debug state entry on System registers, the Event register, and Exclusives monitors

Entering Debug state has no effect on System registers other than [DLR](#) and [DSPSR](#). In particular, ESRs, FARs, and FSRs are not updated on entering Debug state.

Entering Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors.

Entry to Debug state might set the Event Register or clear the Exclusives monitors, or both. However, this is not a requirement, and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

Unless otherwise described in this reference manual, instructions executed in Debug state have their architecturally-defined effects on the System registers, the Event register, and Exclusives monitors.

E2.3.4 Effect of entering Debug state on PSTATE

The effect of an entry to Debug state on PSTATE is described in [Entering Debug state](#) and [Entering Debug state from AArch32 state](#).

On entry to Debug state after PSTATE is saved in [DSPSR](#):

For more information, see [PSTATE in Debug state](#).

E2.3.5 Entering Debug state during loads and stores

The PE can enter Debug state during instructions that perform a sequence of memory accesses, as opposed to a single single-copy atomic access, because of a Watchpoint debug event. The effect of entering Debug state on such an instruction is the same as taking a Data Abort exception during such an instruction.

In addition, when executing in AArch64 state, the PE can enter Debug state because of an External Debug Request debug event during instructions that perform a sequence of memory accesses. The effect of entering Debug state on such an instruction is the same as taking an interrupt exception during such an instruction.

This applies to all memory types.

E2.3.6 Entering Debug state and Software Step

When Software Step is active, a debug event that causes entry to Debug state behaves like an exception taken to an Exception level above the debug target Exception level. That is:

- If the instruction that is stepped generates a synchronous debug event that causes entry to Debug state, or an asynchronous debug event is taken before the step completes, the PE enters Debug state with **DSPSR**.SS set to 1.
- A pending Halting debug event or an asynchronous debug event can be taken after the step has completed. In this case the PE enters Debug state with **DSPSR**.SS set to 0.

In addition:

- If the instruction that is stepped generates an exception trapped by an Exception Catch debug event, the PE enters Debug state at the exception vector with **DSPSR**.SS set to 0. This is because PSTATE.SS is set to 0 by taking the exception.
- If the PE is reset, PSTATE.SS is reset to 0. If the following debug events are enabled, and halting is allowed, the PE enters Debug state with **DSPSR**.SS set to 0:
 - Reset Catch debug event at the reset Exception level.
 - Exception Catch debug event at the reset Exception level.
 - Halting Step debug event.
- If Halting Step is also active, then Halting Step and Software Step operate in parallel and can both become active-pending. In this case Halting step has a higher priority than Software step. This means that the PE enters Debug state and **DSPSR**.SS is set to 0.

E2.3.7 Pseudocode description of entering Debug state

The `DebugHalt` constants are described in [shared/debug/halting/DebugHalt](#) in the pseudocode. The `UpdateEDSCRFields()` and `Halt()` functions are described in [R-profile Architecture Pseudocode](#).

E2.4 Behavior in Debug state

Instructions are executed in Debug state from the Instruction Transfer Register, ITR. The debugger controls which instructions are executed in Debug state by writing the instructions to the External Debug Instruction Transfer register, [EDITR](#).

The Execution state of the PE determines which instruction set is executed:

- If the PE is in AArch32 state it executes T32 instructions:
 - For a 32-bit T32 instruction, [EDITR](#)[15:0] specifies the first halfword and [EDITR](#)[31:16] specifies the second halfword.
 - For a 16-bit T32 instruction, [EDITR](#)[15:0] contains the instruction and [EDITR](#)[31:16] is ignored. All 16-bit T32 instructions are UNPREDICTABLE in Debug state.

The PE does not execute A32 instructions in Debug state.

Some instructions are available only in Debug state. See [Debug state operations](#), [DCPS](#), [DRPS](#), [MRS](#), [MSR](#). In Non-debug state these instructions are UNDEFINED.

The following sections describe behavior in Debug state:

- [PSTATE in Debug state](#).
- [Executing instructions in Debug state](#).
- [Decode tables](#).
- [Security in Debug state](#).
- [Privilege in Debug state](#).
- [Debug state operations](#), [DCPS](#), [DRPS](#), [MRS](#), [MSR](#).
- [Exceptions in Debug state](#).
- [Accessing registers in Debug state](#).
- [Accessing memory in Debug state](#).

This section specifies the CONSTRAINED UNPREDICTABLE behaviors that apply in Debug state, but see [Changing the value of EDECR.SS when not in Debug state](#) for a change in Non-debug state that causes CONSTRAINED UNPREDICTABLE behavior.

E2.4.1 PSTATE in Debug state

PSTATE. are all ignored in Debug state:

- There are no conditional instructions in Debug state.
- In AArch32 state, the PE executes only T32 instructions and [PSTATE.IT](#) is ignored.
- Software step is inactive.
- Asynchronous exceptions and debug events are ignored.

Instructions executed in Debug state indirectly read PSTATE.{TCO, UAO, PAN, IL, E, M, nRW, EL, SP} as they would in Non-debug state.

Note

[PSTATE.DIT](#) is not guaranteed to have any effect in Debug state.

In Debug state:

E2.4.2 Executing instructions in Debug state

The instructions executed in Debug state must be either A64 instructions or T32 instructions, depending on the current Execution state.

Each instruction falls into one of the following groups:

- Debug state instructions. These are instructions that are changed in Debug state. See A64 instructions that are changed in Debug state and [T32 instructions that are changed in Debug state](#).
- Instructions that are unchanged in Debug state. See A64 instructions that are unchanged in Debug state and [T32 instructions that are unchanged in Debug state](#). Where these sections describe the behavior of an instruction as unchanged in Debug state, or having the same behavior as in Non-debug state, and that instruction would generate an exception in Non-debug state, then the behavior of the exception is as described in [Exceptions in Debug state](#).
- Instructions that are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state. See A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state and [T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state](#).

All T32 instructions are treated as unconditional, regardless of [PSTATE.IT](#). See [PSTATE in Debug state](#).

E2.4.2.1 Executing T32 instructions in Debug state

The following sections describe the behavior of the T32 instructions in Debug state:

- [T32 instructions that are changed in Debug state](#).
- [T32 instructions that are unchanged in Debug state](#).
- [T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state](#).

E2.4.2.1.1 T32 instructions that are changed in Debug state

The following T32 instructions are defined in Debug state, but are UNDEFINED in Non-debug state:

- DCPS.

Note

DCPS can be UNDEFINED in certain conditions in Debug state. See [DCPS<n>](#).

- MRC p15, 3, <Rt>, c4, c5, 0 (DSPSR).
- MCR p15, 3, <Rt>, c4, c5, 0 (DSPSR).
- MRC p15, 3, <Rt>, c4, c5, 1 (DLR).
- MCR p15, 3, <Rt>, c4, c5, 1 (DLR).

In addition, [ERET](#) executes the DRPS operation in Debug state.

For more information, see [Debug state operations](#), [DCPS](#), [DRPS](#), [MRS](#), [MSR](#).

E2.4.2.1.2 T32 instructions that are unchanged in Debug state

The following list shows the instructions that are unchanged in Debug state. Any T32 instruction that uses the PC or APSR. {N, Z, C, V} as the source or destination register is not included in the list. Moreover, the list includes only the 32-bit T32 encodings.

Any instruction that is UNDEFINED in Non-debug state

The list of instructions:

- Excludes any instruction listed in [T32 instructions that are changed in Debug state](#).
- Excludes any instruction listed in [T32 instructions that are CONSTRAINED UNPREDICTABLE in Debug state](#) that is UNDEFINED because an enable or disable bit is not RES0 or RES1.

Instructions that move System or Special-purpose registers to or from a general-purpose register

The list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.
- Excludes APSR and CPSR access instructions.
- Excludes instructions for accessing banked registers for the current mode.

These instructions are:

- MRS <banked_reg>, MSR <banked_reg>.

Note

This does not apply to cases which are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state in the current mode.

- MRC, MCR.

Note

This includes all allocated System registers in the (coproc == 0b111x) encoding space other than an MRC move to APSR_nzcv.

- MRS SPSR, MSR SPSR_fsrc (register).
- VMRS <vfp_system_reg>, VMSR <vfp_system_reg>.

Note

This includes all allocated Advanced SIMD and floating-point System registers, other than an a VMRS move to APSR_nzcv.

Floating-point moves between a SIMD&FP register and a general-purpose register

These instructions are:

- VMOV (between a general-purpose register and a single-precision register).
- VMOV (between a general-purpose register and a doubleword floating-point register).

SIMD moves between a SIMD&FP register and a general-purpose register

These instructions are:

- VMOV (between a general-purpose register and a scalar).

Barriers

These instructions are:

- DFB
- DMB.
- DSB.
- ISB.

Memory access instructions at various access sizes

The following constraints apply:

- General purpose-registers only.
- One of the following addressing modes:
 - Immediate (8-bit or 12-bit) offset.
 - Immediate (8-bit) post-indexed.
 - Immediate (8-bit) pre-indexed.
 - Unprivileged (8-bit).
- Not literal.
- One of the following types:
 - (Single) register.
 - Dual.
 - Exclusive.
 - Exclusive doubleword.
 - Acquire/Release.
 - Acquire/Release Exclusive.
 - Acquire/Release Exclusive doubleword.

These instructions are:

- LDR.W, LDRB.W, LDRH.W, LDRD, LDRSB.W, LDRSH.W (immediate, not literal).
- LDRT, LDRBT, LDRHT, LDRSBT, LDRSHT(immediate).
- LDREX, LDREXB, LDREXH, LDA, LDAB, LDAH, LDAEX, LDAEXB, LDAEXH.
- LDREXD, LDAEXD.
- STR.W, STRB.W, STRH.W, STRD (immediate).
- STRT, STRBT, STRHT (immediate).
- STREX, STREXB, STREXH, STL, STLB, STLH, STLEX, STLEXB, STLEXH.
- STREXD, STLEXD.

Move to general-purpose register

These instructions are:

- MOVW, MOVT(immediate).

System instructions, Send Event, NOP, and Clear Exclusive

The System instructions are Cache maintenance instructions, TLB maintenance instructions, and Address translation instructions. These are encoded in the (coproc == 0b1111) System register encoding space.

These instructions are:

- ICIALLU, ICIALUIS, ICIMVAU.
- DCCIMVAC, DCCISW, DCCMVAC, DCCMVAU, DCCSW, DCIMVAC, DCISW.
- TLBIALL, TLBIALLH, TLBIALLHIS, TLBIALLIS, TLBIALLNSNH, TLBIALLNSNHIS, TLBIASID, TLBIASIDIS, TLBIIPAS2, TLBIIPAS2IS, TLBIIPAS2L, TLBIIPAS2LIS, TLBIMVA, TLBIMVAA, TLBIMVAAIS, TLBIMVAAL, TLBIMVAALIS, TLBIMVAH, TLBIMVAHIS, TLBIMVAIS, TLBIMVAL, TLBIMVALH, TLBIMVALHIS, TLBIMVALIS.
- ATS12NSOPR, ATS12NSOPW, ATS12NSOUR, ATS12NSOUW, ATS1CPR, ATS1CPW, ATS1CUR, ATS1CUW, ATS1HR, ATS1HW.
- BPIALL, BPIALLIS, BPIMVA.
- SEV.W, SEVL.W.
- NOP.W.
- CLREX.

E2.4.2.1.3 T32 instructions that are **CONSTRAINED UNPREDICTABLE** in Debug state

This subsection describes all instruction not listed in either:

- [T32 instructions that are changed in Debug state.](#)
- [T32 instructions that are unchanged in Debug state.](#)

These instructions are **CONSTRAINED UNPREDICTABLE** in Debug state. In general, the permissible behaviors are:

- The instruction generates an Undefined Instruction exception.
- The instruction executes as a NOP.
- If the instruction reads the PC or [PSTATE](#), it uses an UNKNOWN value.
- If the instruction modifies the PC or [PSTATE](#), other than by advancing the PC to the sequentially next instruction, it sets [DLR](#), and [DSPSR](#) to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for T32 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed.

Exception-generating instructions

These instructions are:

- SVC.

- HVC.
- SMC.
- UDF.
- BKPT.
- HLT.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- SVC behaves as DCPS1.
- HVC behaves as DCPS2.
- SMC behaves as DCPS3.
- They generate the exception the instruction would generate in Non-debug state. The exception is taken as described in [Exceptions in Debug state](#).

Note

SMC must not generate a Secure Monitor Call exception if EDSCR.SDD is 1.

Instructions that explicitly write to the PC (branches)

These instructions are:

- B, B (conditional), CBZ, CBNZ, BL.
- BX, BLX (register or immediate).
- BXJ, TBB, TBH.
- MOV pc and related instructions.
- LDR pc, LDM (with a register list includes the PC), POP (with a register list that includes the PC).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching and set [DLR](#) and [DSPSR](#) to UNKNOWN values.

Exception return and related instructions, other than ERET

These instructions are:

- SRS, RFE, SUBS pc, 1r, and related instructions.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state without branching, setting [DLR](#) and [DSPSR](#) to UNKNOWN values, and either:
 - Execute the DRPS operation instead of performing an exception return, using UNKNOWN SPSR values.
 - Not changing Exception level or PE mode.

Instructions that request entry to a low-power state

These instructions are:

- WFE, WFI, WFET, WFIT

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.

- They generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See [Configurable instruction controls](#).
- A WFE instruction is permitted to clear the Event register if it is set.

Note

This means that these instructions must not suspend execution.

Instructions that read the PC

These instructions are:

- LDR (literal), LDRB (literal), LDRH (literal), LDRSB (literal), LDRSH (literal).
- ADR, ADRL, ADRH.
- PLD (literal), PLI (literal).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state using an UNKNOWN value for the PC operand.

Instructions that explicitly modify **PSTATE**, other than DCPS and ERET

These instructions are:

- CMP, TST, TEQ, CMN.
- <opc>S.
- MRC p14, 0, APSR_nzcv, c0, c1, 0 (accessing DBGDSCRint).
- CPS, SETEND, IT.
- MSR CPSR (immediate), MSR CPSR (register), MSR APSR (immediate), MSR APSR (register).
- VMRS APSR_nzcv, FPSCR.
- QADD, QDADD, QSUB, QDSUB.
- SMLABB, SMLABT, SMLATB, SMLATT, SMLAD, SMLAWB, SMLAWT, SMLSD, SMUAD.
- SSAT, SSAT16, USAT, USAT16.
- SADD, SADD8, SADD16, SASX, SSAX, SSUB, SSUB8, SSUB16.
- UADD, UADD8, UADD16, UASX, USAX, USAUB, USUN8, USUB16.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state, setting **DLR**, **DSPSR**, and any PSTATE field written by the instruction to UNKNOWN values.

Instructions that read **PSTATE**.{N, Z, C, V} or other **PSTATE** fields

These instructions are:

- SEL, VSEL.
- ADC, SBC, all instructions with an RRX shift.
- MRS CPSR “.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state:
 - For the conditional operations and those using the **PSTATE.C** flag as an input, these instructions use an UNKNOWN value for the Condition flag.
 - For the MRS instruction, they return an UNKNOWN value.

All other instructions

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They have the same behavior as in Non-debug state.

Note

This includes instructions defined as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state. These instructions are CONSTRAINED UNPREDICTABLE in Debug state. This includes some T32 instructions that specify R15 as a destination or source register.

[Architectural Constraints on UNPREDICTABLE Behaviors](#) describes the CONSTRAINED UNPREDICTABLE behavior for these instructions. In Debug state these CONSTRAINED UNPREDICTABLE choices are further restricted:

- Instructions that specify R15 as a destination register:
 - Are not permitted to branch, because the architecture does not define a branch operation in Debug state.
 - Might set [DLR](#) and [DPSR](#) to UNKNOWN values.
 - Might have any of the other permitted behaviors.
 - Instructions that specify R15 as a source operand:
 - Cannot use PC + offset, because there is no architecturally-defined PC in Debug state.
 - Might have any of the other permitted behaviors, including using an UNKNOWN value.
-

E2.4.3 Decode tables

The syntax in the decode tables is defined as follows:

1	The bit has a fixed value of 1.
0	The bit has a fixed value of 0.
!=	The field has any value other than the value or values specified. The field might be an encoding field in the instruction whose value is supplied by the debugger.

Note

The instruction encodings in A64 Base Instruction Descriptions and [T32 and A32 Base Instruction Set Instruction Descriptions](#) might show these bits as (0) or (1). A debugger must set these bits to 0 or 1, as appropriate.

Any other value indicates an encoding field in the instruction whose value is supplied by the debugger. Some values might be reserved or UNDEFINED, in which case the instruction is UNDEFINED or CONSTRAINED UNPREDICTABLE in Debug state, as it is in Non-debug state.

For more information about the instruction encodings, see [T32 and A32 Base Instruction Set Instruction Descriptions](#).

For information about the syntax used in [Figure E2-1](#) and [Figure E2-2](#) see:

- Common syntax terms.
- [Assembler symbols](#).

E2.4.3.1 T32 instructions that are modified in Debug state

[Figure E2-1](#) shows the T32 instructions that are modified in Debug state, with the first halfword on the left side and the second halfword on the right side. For information of how these are packed in the [EDITR](#), see the register description.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 1 0	1 1 1 0	0 1 1 L	0 1 0 0	!=1111	1 1 1 1	0 0 0 1	0 1 0 1	MRC MCR accessing DSPSR
1 1 1 0	1 1 1 0	0 1 1 L	0 1 0 0	!=1111	1 1 1 1	0 0 1 1	0 1 0 1	MRC MCR accessing DLR
1 1 1 1	0 0 1 1	1 1 0 1	1 1 1 0	1 0 0 0	1 1 1 1	0 0 0 0	0 0 0 0	ERET
1 1 1 1	0 1 1 1	1 0 0 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 !=00	DCPS<opt>

Figure E2-1 Modified T32 instructions in Debug state

E2.4.3.2 T32 instructions that are unchanged in Debug state

Figure E2-2 shows the T32 instructions that are unchanged in Debug state, other than some unallocated and UNDEFINED instructions. It shows the T32 instructions with the first halfword on the left side and the second halfword on the right side.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description	
1 1 1 0	1 1 0 0	0 1 0	op	!=1111	1 0 1 1	0 0 M 1	Vm	VMOV <Dm>, <Rt>, <Rt2> VMOV <Rt>, <Rt2>, <Dm>	
1 1 1 0	1 1 1 0	0 0 0	op	Vn	1 0 1 0	N 0 0 1	0 0 0 0	VMOV <Sn>, <Rt>, VMOV <Rt>, <Sn>	
1 1 1 0	1 1 1 0	0	opc	0	Vd	1 0 1 1	Dopc2 1	0 0 0 0	VMOV.<size> <Dd>[<x>], <Rt>
1 1 1 0	1 1 1 0	U	opc	1	Vn	1 0 1 1	Dopc2 1	0 0 0 0	VMOV.<dt> <Rt>, <Dd>[<x>]
1 1 1 0	1 1 1 0	1 1 1	op	reg	!=1111	1 0 1 0	0 0 0 1	0 0 0 0	VMRS, VMSR
1 1 1 0	1 1 0 0	0 1 0	op	!=1111	!=1111	1 1 1	cp	opc1	CRm MCR MRRC accessing System registers
1 1 1 0	1 1 1 0	opc1	op	CRn	!=1111	1 1 1	cp	opc2 1	CRm MCR MRC accessing System registers
1 1 1 0	1 0 0 0	0 1 0	L	!=1111	!=1111	Rd		imm8	LDREX, STREX
1 1 1 0	1 0 0 0	1 1 0	L	!=1111	!=1111	Rt2	0 1	!=10 Rd	LDREX(B H D), STREX(B H D)
1 1 1 0	1 0 0 0	1 1 0	L	!=1111	!=1111	Rt2	1	op3 Rd	LDA{EX}{B H D}, STL{EX}{B H D}
1 1 1 0	1 0 0	!=0x10 !=xx0x	L	!=1111	!=1111	!=1111		imm8	LDRD, STRD
1 1 1 1	0 i 1 0	T 1 0 0		imm4	0	imm3	!=1111	imm8	MOVW, MOVT
1 1 1 1	0 0 1 1	1 0 0	R	!=1111	1 0 0 0	M1	0 0 1	M 0 0 0	MSR <spec_reg><mode>, <Rn>
1 1 1 1	0 0 1 1	1 0 0 1		!=1111	1 0 0 0	1 1 1 1	0 0 0 0	0 0 0 0	MSR SPSR, <Rn>
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	NOP.W
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 1 0	L SEV.W, SEVL.W
1 1 1 1	0 0 1 1	1 0 1 0	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	op00	ESB, CSDB
1 1 1 1	0 0 1 1	1 0 1 1	1 1 1 1	1 0 0 0	1 1 1 1	0 0 1 0	1 1 1 1		CLREX
1 1 1 1	0 0 1 1	1 0 1 1	1 1 1 1	1 0 0 0	1 1 1 1	0 1	op	option	DSB, DMB, ISB, SSBB, PSSBB, SB
1 1 1 1	0 0 1 1	1 1 1	R	M1	1 0 0 0	!=1111	0 0 1	M 0 0 0	MRS <Rd>, <spec_reg><mode>
1 1 1 1	0 0 1 1	1 1 1 1	1 1 1 1	1 0 0 0	!=1111	0 0 0 0	0 0 0 0	0 0 0 0	MRS <Rd>, SPSR
1 1 1 1	1 0 0 0	1	!=11	0	!=1111	!=1111		imm12	STR{B H}.W (12-bit immediate)
1 1 1 1	1 0 0 0	0	!=11	0	!=1111	!=1111	1	!=000	imm8 STR{B H}{T} (8-bit immediate)
1 1 1 1	1 0 0	S 1	!=11	1	!=1111	!=1111		imm12	LDR{SB SH B H}.W (12-bit immediate)
1 1 1 1	1 0 0	S 0	!=11	1	!=1111	!=1111	1	!=000	imm8 LDR{SB SH B H}{T} (8-bit immediate)

Figure E2-2 T32 instructions that are unchanged in Debug state

E2.4.4 Security in Debug state

On entry to Debug state

[EDSCR.SDD](#) is the Secure debug disabled flag. If entering Debug state from Non-secure state, [EDSCR.SDD](#) is set to 1.

Note

If halting is prohibited in a Security state, it is not possible to enter Debug state from that Security state. However, because changes to the authentication signals require a [Context Synchronization event](#) to guarantee their effect, there is a period during which the PE might halt even though the authentication signals prohibit halting.

In Debug state

The value of [EDSCR.SDD](#) does not change, even if [ExternalSecureInvasiveDebugEnabled\(\)](#) changes.

Note

- [DBGAUTHSTATUS_EL1](#) is not frozen in Debug state. However, a Context Synchronization event is required to guarantee that changes are visible in [DBGAUTHSTATUS_EL1](#).
 - If [EDSCR.SDD](#) is 1 in Debug state, there is no means to enter a Security state where halting was prohibited on entry to Debug state. This means that it is not possible for the PE to be in such a Security state when [EDSCR.SDD](#) is 1. This is a general principle of behavior in Debug state.
-

In Non-debug state

[EDSCR.SDD](#) returns the inverse of [ExternalSecureInvasiveDebugEnabled\(\)](#). If the authentication signals that control [ExternalSecureInvasiveDebugEnabled\(\)](#) change, a [Context Synchronization event](#) is required to guarantee their effect.

Note

- A [Context Synchronization event](#) is also required to guarantee that changes in the authentication signals are visible in [DBGAUTHSTATUS_EL1](#).
-

E2.4.5 Privilege in Debug state

The only additional privileges offered to Debug state are:

- The privilege to execute [Debug state operations, DCPS, DRPS, MRS, MSR](#).
- The privilege to execute DTR access instructions regardless of the Exception level and traps.

The DTR access instructions can be executed at any Exception level, including EL0, regardless of any control register settings that might force these instructions to be UNDEFINED or trapped in Non-debug state. These instruction are:

- The MRC and MCR instructions that access [DBGDTRTXint](#) and [DBGDTRRXint](#) in AArch32 state.

All other instructions operate with the privilege determined by the current Exception level and Security state. This applies to all Special-purpose and System registers accesses, memory accesses, and UNDEFINED instructions, and includes generating exceptions when the System registers trap or disable an instruction.

E2.4.6 Debug state operations, DCPS, DRPS, MRS, MSR

The architecture defines operations to change between Exception levels in Debug state. These operations can also change the mode at the current Exception level.

E2.4.6.1 DCPS<n>

Executing a DCPS<n> instruction in Debug state moves the PE to a higher Exception level or to a specific mode at the current Exception level.

If the target Exception level is the same as the current Exception level, then the PE does not change Exception level. However, the PE might change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state the current endianness is determined by the value of [SCTLR.EE](#) or [HSCTLR.EE](#) for the target Exception level.

In AArch32 state, in the T32 instruction set only

- DCPS1
- DCPS2
- DCPS3

The DCPS instructions are UNDEFINED in Non-debug state.

[Table E2-2](#) shows the target of the instruction. In [Table E2-2](#), the entries have the following meaning:

EL1h/Svc This means that the target is EL1 and Supervisor mode.

EL2h/Hyp This means that the target is EL2 and Hyp mode.

Table E2-2 Target for DCPS instructions in Debug state

Instruction	Target when DCPS instruction executed at stated Exception level:		
	EL0	EL1	EL2
DCPS1	EL1h	EL1h	EL2h
DCPS2	EL2h	EL2h	EL2h
DCPS3	•	•	•

However:

- DCPS1 is UNDEFINED at EL0 if HCR_EL2.TGE == 1.
- DCPS3 is UNDEFINED at all Exception levels.

On executing a DCPS instruction:

- If the target Exception level is using AArch32, [DLR](#), [DSPSR](#), and if FEAT_Debugv8p9 is implemented, DSPSR2, become UNKNOWN and:
 - If the target Exception level is EL1 the LR and SPSR of the target mode become UNKNOWN.
 - If the target Exception level is EL2, then [ELR_hyp](#), [SPSR_hyp](#), and [HSR](#) become UNKNOWN.

If the target Exception level is using AArch32, and the target Exception level is EL1 the LR and SPSR of the target mode become UNKNOWN.

The [DCPSInstruction\(\)](#) function is described in [A-profile Architecture Pseudocode](#).

E2.4.6.2 DRPS

Executing the DRPS operation in Debug state moves the PE to a lower Exception level, or the current Exception level, by copying the current SPSR to PSTATE.

If DRPS is executed in AArch64 state and the target Exception level is using AArch32:

- The current instruction set switches from A64 to T32.
- The effect on registers that are not visible or only partially visible in AArch32 state is the same as for exception returns in Non-debug state. See Execution state.

Otherwise, the instruction set state does not change.

If the target Exception level is the current Exception level, then the PE does not change Exception level. However, the PE might change mode.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state:

- If targeting an Exception level using AArch64, current endianness is set according to SCTLR_ELx.EE, or SCTLR_EL1.E0E for the target Exception level.
- If targeting an Exception level using AArch32, current endianness is set by SPSR.E as appropriate.

The DRPS instructions are:

In AArch32 state, in the T32 instruction set only

- `ERET`

If the SPSR specifies an illegal exception return, then PSTATE.{M, nRW, EL, SP} are unchanged and PSTATE.IL is set to 1. For further information on illegal exception returns, see Illegal exception returns from AArch64 state.

Some PSTATE fields are ignored in Debug state. The effect of the DRPS operation on them is to set them to an UNKNOWN value that might be the value from the SPSR. For more information, see [PSTATE in Debug state](#).

All other PSTATE fields are copied from SPSR.

Unlike an exception return, the DRPS operation has no architecturally-defined effect on the Event Register and Exclusives monitors. DRPS might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

On executing a DRPS instruction then [DLR](#) and [DSPSR](#) become UNKNOWN.

The [DRPSInstruction\(\)](#) function is described in [R-profile Architecture Pseudocode](#).

E2.4.6.3 MRS and MSR

The other Debug state instructions are used to read or write DLR_EL0 and DSPSR_EL0.

These instructions are:

In AArch32 state

- [MRC](#)
- [MCR](#)

MRC <Rn>, DLR	; Copy DLR to <Rn>
MRC <Rn>, DSPSR	; Copy DSPSR to <Rn>
MCR DLR, <Rn>	; Copy <Rn> to DLR
MCR DSPSR, <Rn>	; Copy <Rn> to DSPSR

These instructions can be executed at any Exception level when in Debug state, including EL0. They are UNDEFINED in Non-debug state.

E2.4.7 Exceptions in Debug state

The following sections describe how exceptions are handled in Debug state:

- [Generating exceptions when in Debug state.](#)
- [Taking exceptions when in Debug state.](#)
- [Reset in Debug state.](#)

E2.4.7.1 Generating exceptions when in Debug state

In Debug state:

- Instruction Abort exceptions cannot happen because instructions are not fetched from memory.
- Interrupts, including SError and virtual interrupts are ignored and remain pending:
 - The pending interrupt remains visible in [ISR](#).
- Debug exceptions and debug events are ignored.
- Any attempt to execute an instruction bit pattern that is an allocated instruction at the current Exception level, but is listed in [Executing instructions in Debug state](#) as UNDEFINED in Debug state, generates an exception. The priority and syndrome for these exceptions is the same as for executing an encoding that does not have an allocated instruction.

Otherwise, synchronous exceptions are generated as they would be in Non-debug state and taken to the appropriate Exception level in Debug state.

E2.4.7.2 Taking exceptions when in Debug state

When the PE is in Debug state, all exceptions are synchronous. When an exception is generated, it is taken to Debug state. This means that:

- The target Exception level is as defined for the exception in Non-debug state.
- The exception syndrome is reported as defined for the exception in Non-debug state, except for the case described in [Data Aborts in Memory access mode](#) for which the reporting requirements are relaxed. The exception syndrome is reported using the syndrome register or registers for the target Exception level. In AArch64 state, these are [DFSR](#), [DFAR](#), [HSR](#), [HDFAR](#), and [HPFAR](#). For example, if an instruction is trapped to an Exception level using AArch64 due to a configurable trap, disable, or enable, the exception code reported is the same as it would be in Non-debug state.

Note

Generally, the AArch32 Fault Address Registers (FARs) and Fault Status Registers (FSRs) are not described as *syndrome registers*, although the term is appropriate to their function.

The effect on auxiliary syndrome registers, such as AFSR, is IMPLEMENTATION DEFINED.

- The PE remains in Debug state and changes to the target mode.
- The endianness switches to that indicated by the EE bit of the System register for the target Exception level.
- The SPSR for the target Exception level or mode is corrupted and becomes UNKNOWN.
- If the target Exception level is using AArch64, ELR_ELx for the target Exception level becomes UNKNOWN.
- [DLR](#) and [DPSR](#) become UNKNOWN.
- The cumulative error flag, [EDSCR.ERR](#), is set to 1. See [Cumulative error flag](#).
- PSTATE.IL is cleared to 0.
- PSTATE.{IT, T, SS, D, A, I, F} are set to UNKNOWN values, and PSTATE.{N, Z, C, V, Q, GE} are unchanged. However, these fields are ignored and are not observable in Debug state. For more information, see [PSTATE in Debug state](#).

The debugger must save any state that can be corrupted by an exception before executing an instruction that might generate another exception.

E2.4.7.3 Pseudocode description of taking exceptions in Debug state

[AArch32_EnterMode\(\)](#), [AArch32_EnterHypMode\(\)](#), and [AArch32_EnterMonitorMode\(\)](#) show the behavior when the PE takes an exception to an Exception level using AArch32 in Non-debug state. In Debug state:

- [AArch32_EnterMode\(\)](#) redirects to [AArch32_EnterModeInDebugState\(\)](#).
- [AArch32_EnterHypMode\(\)](#) redirects to [AArch32_EnterHypModeInDebugState\(\)](#).
- [AArch32_EnterMonitorMode\(\)](#) redirects to [AArch32_EnterMonitorModeInDebugState\(\)](#).

E2.4.7.4 Reset in Debug state

If the PE is reset when in Debug state, it exits Debug state and enters Non-debug reset state. When the PE is in reset state, [EDSCR.STATUS](#) == 0b000010 and writes to [EDITR](#) are ignored.

Note

If [EDECR.RCE](#) == 1 or [CTIDEVCTL.RCE](#) == 1, meaning that a Reset Catch debug event is programmed, and if halting is allowed on exiting reset state, then on exiting reset state the PE halts and reenters Debug state. See [Reset Catch debug events](#). All PE registers have taken their reset values, which might be UNKNOWN.

E2.4.8 Accessing registers in Debug state

Register accesses are unchanged in Debug state. The view of each register is determined by either the current Exception level or the mode, or both, and accesses might be disabled or trapped by controls at a higher Exception level.

E2.4.8.1 General-purpose register access

A single general-purpose register can be read by issuing an `nMCR` instruction through the ITR to write `DBGDTRTXint`.

The debugger can then read the DTR register or registers through the external debug interface. The reverse sequence writes to a general-purpose register.

Figure E2-3 shows the reading and writing of general-purpose registers in Debug state in AArch32 state.

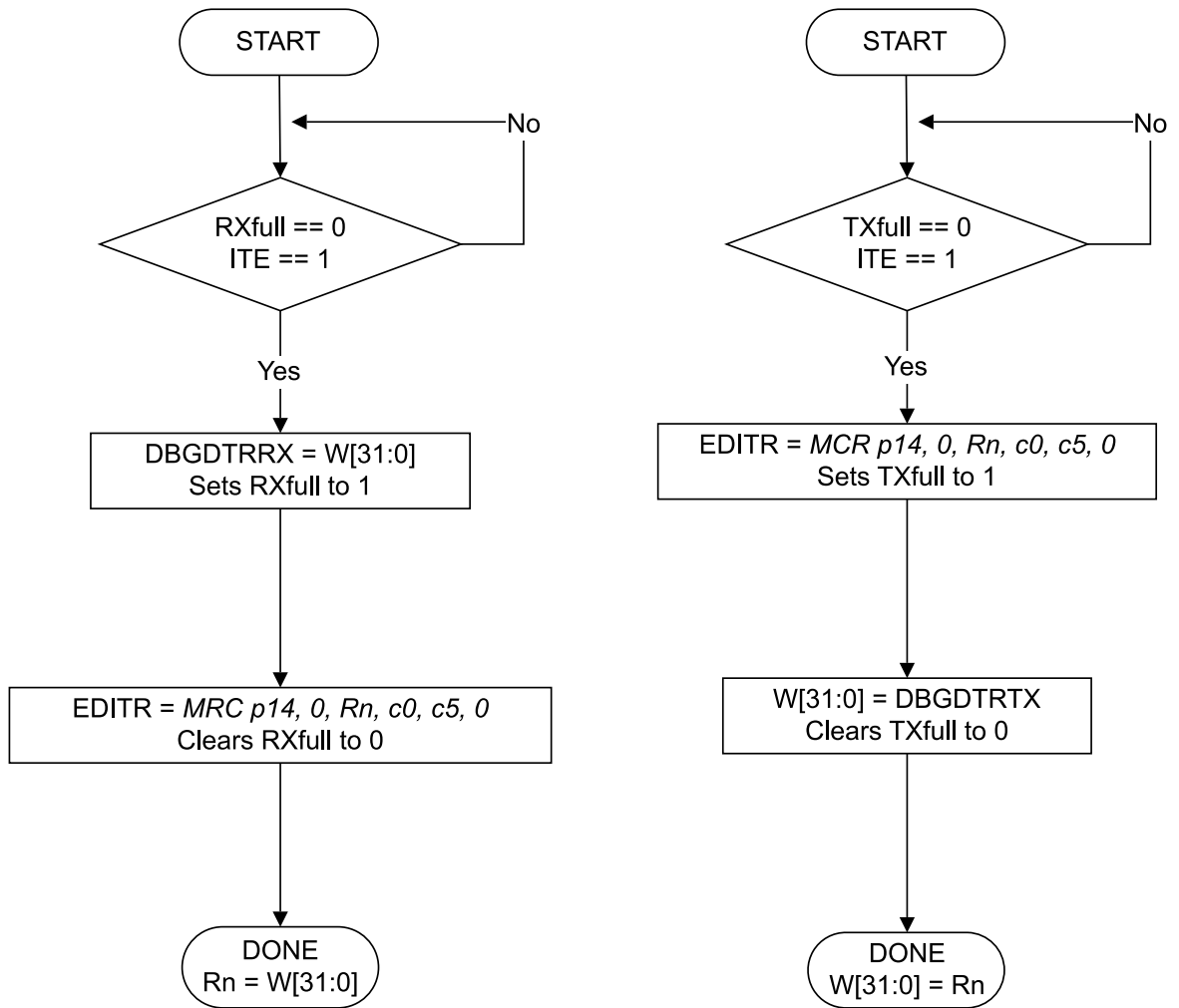


Figure E2-3 Reading and writing general-purpose registers in Debug state in AArch32 state

E2.4.8.2 PC and PSTATE access

The debugger reads the Program Counter and PSTATE of the process being debugged through the `DLR` and `DSPSR` System registers. The actual values of PC and PSTATE cannot be directly observed in Debug state:

- Instructions that are used for direct reads and writes of PC and PSTATE in Non-debug state are **CONSTRAINED UNPREDICTABLE** in Debug state.
- On taking an exception, `ELR_ELx` and `SPSR_ELx` at the target Exception level are **UNKNOWN**. They do not record the PC and PSTATE.

PSTATE. are indirectly read by instructions executed in Debug state, but all other PSTATE fields are ignored and cannot be observed. See also:

- [PSTATE in Debug state.](#)
- [Executing instructions in Debug state.](#)
- [Exceptions in Debug state.](#)

E2.4.8.3 Other register accesses

To access other registers, the debugger can use the operations described in [General-purpose register access](#) and the operations described below to build a sequence of operations to move the registers to or from a general-purpose register.

To access a SIMD&FP, SP, or System register, the debugger can use an instruction from the following list to transfer the register to or from a general-purpose register:

- The `VMOV` instruction in AArch32 state for SIMD transfers from SIMD&FP registers.
- The `VMOV` instruction in AArch32 state for floating-point transfers to or from SIMD&FP registers.
- [MCR](#), [MCRR](#), [MRRC](#), and [MRC](#) instructions in AArch32 state for transfers to or from System registers.

E2.4.9 Accessing memory in Debug state

How the PE accesses memory is unchanged in Debug state. This includes:

- The operation of the MPU, including address translation, access permissions, and memory attribute determination.
- The operation of any caches and coherency mechanisms.
- Alignment support.
- Endianness support.
- The Memory order model.
- The operation of the MPU, including access permissions and memory attribute determination.

E2.4.9.1 Simple memory transfers

Simple memory accesses can be performed in Debug state by issuing memory access instructions through the ITR and passing data through the DTR registers. [Executing instructions in Debug state](#) lists the memory access instructions that are supported in Debug state.

E2.4.9.2 Bulk memory transfers

Memory access mode can accelerate bulk memory transfers in Debug state. See [DCC and ITR access modes](#).

E2.5 Exiting Debug state

The PE exits Debug state when it receives a Restart request trigger event. If `EDSCR.ITE == 0`, when the PE receives a Restart Request trigger event, the behavior of any instruction issued through the ITR in Normal access mode or an operation issued by a DTR access in Memory access mode that has not completed execution is one of the following:

- The instruction completes execution in Debug state and the PE ignores the restart request and remains in Debug state.
- The instruction completes execution in Debug state before the PE executes the restart sequence.
- The instruction completes execution in Non-debug state after the PE executes the restart sequence.
- The instruction is abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

Note

- Implementations can set `EDSCR.ITE` to 1 to indicate that further instructions can be accepted by ITR before the previous instructions have completed. If any previous instruction has not completed and `EDSCR.ITE == 1`, then the PE must complete these instructions in Debug state before executing the restart sequence. `EDSCR.ITE == 0` indicates that the PE is not ready to restart.
 - A debugger must observe that any instructions issued through `EDITR` that might generate a synchronous exception, as complete, before issuing a restart request. It can do this by observing the completion of a later instruction, as synchronous exceptions must occur in program order. For example, a debugger can observe that an instruction that reads or writes a DTR register is complete because of its effect on the `EDSCR.{TXfull, RXfull}` flags.
-

On exiting Debug state, the PE sets the Program Counter to the address in `DLR`, where:

- If exiting to AArch32 state:
 - Bits[31:1] of the PC are set to the value of bits[31:1] of `DLR`.
 - Bit[0] of the PC is set to a CONSTRAINED UNPREDICTABLE choice of 0 or the value of bit[0] in `DLR`.

Exit from Debug state can give rise to a PC alignment fault exception when the Program Counter is used. Unlike an exception return, this might also happen when returning to AArch32 state. For more information, see PC alignment checking.

On exiting Debug state, `PSTATE` is set from `DSPSR` in the same way that an exception return sets `PSTATE` from `SPSR_ELx`:

- The same illegal exception return checks that apply to an exception return also apply to exiting Debug state. If the return from Debug state is an illegal exception return then the effect on `PSTATE` and the PC is the same as for any other illegal exception return. See Exception return and [Exception return to an Exception level using AArch32](#).
- The checks on the `PSTATE.IT` bits that apply to exiting Debug state into AArch32 state are the same as those that apply to an exception return. See [Architectural Constraints on UNPREDICTABLE Behaviors](#).
- `PSTATE.SS` is copied from `DSPSR.SS` if all of the following hold:
 - Software step exceptions from the restart Exception level are enabled.Otherwise `PSTATE.SS` is set to 0.

Note

Unlike a return using `ERET`, `PSTATE.SS` must be restored from `DSPSR.SS` because otherwise it is UNKNOWN.

However, if `DBGOSDLR.DLK == 1` and `DBGPRCR.CORENPDRQ == 0`, meaning `FEAT_DoubleLock` is implemented and locked in Non-debug state and therefore Software Step exceptions are disabled, but otherwise Software Step exceptions would be enabled from the restart Exception level, it is CONSTRAINED UNPREDICTABLE whether `PSTATE.SS` is copied from `DSPSR.SS`.

- Exit from Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors. An exit from Debug state might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

Note

- One important difference between Debug state exit and an exception return is that the PE can exit Debug state at EL0. Despite this, the behavior of an exit from Debug state is similar to an exception return.
-

The `ExitDebugState()` function is described in [R-profile Architecture Pseudocode](#).

Chapter E3

Halting Debug Events

This chapter describes a particular class of debug events. It contains the following sections:

- [Introduction to Halting debug events](#)
- [Halting Step debug events](#)
- [Halt Instruction debug event](#)
- [Exception Catch debug event](#)
- [External Debug Request debug event](#)
- [OS Unlock Catch debug event](#)
- [Reset Catch debug events](#)
- [Software Access debug event](#)
- [Synchronization and Halting debug events](#)

E3.1 Introduction to Halting debug events

External debug defines Halting debug events. The following Halting debug events are available from the introduction of Armv8:

- [Halting Step debug events](#).
- [Halt Instruction debug event](#).
- [Exception Catch debug event](#).
- [External Debug Request debug event](#).
- [OS Unlock Catch debug event](#).
- [Reset Catch debug events](#).
- [Software Access debug event](#).

If halting is allowed, a Halting debug event halts the PE. The PE enters Debug state.

In addition, breakpoints and watchpoints might halt the PE if halting is allowed. See [Breakpoint and Watchpoint debug events](#). Because breakpoints and watchpoints can generate an exception or halt the PE, Breakpoint and Watchpoint debug events are not classified as Halting debug events.

For a definition of Debug state, see [Debug State](#). For a definition of halting allowed, see [Halting allowed and halting prohibited](#).

[Debug state entry and debug event prioritization](#) describes the behavior when multiple debug events are generated by an instruction.

See also [Synchronization and Halting debug events](#).

[Table E3-1](#) shows the behavior of Breakpoint, Watchpoint, and Halting debug events.

Table E3-1 Summary of debug events and possible outcomes

Debug event type	PE behavior when halting is:	
	Allowed	Prohibited
Breakpoint and Watchpoint debug events	Halt	See Table D2-1
Halt Instruction debug event	Halt	UNDEFINED
Software Access debug event	Halt	Ignored
Exception Catch debug event	Halt	Ignored or pended
Halting Step debug events	Halt	Pended
External Debug Request debug event	Halt	Pended
Reset Catch debug events	Halt	Pended
OS Unlock Catch debug event	Pended	Pended

[Table E3-2](#) shows where the pseudocode for each Halting debug event type is located.

Table E3-2 Pseudocode description of Halting debug events

Halting debug event type	Pseudocode
Halt Instruction debug event	HLT

Halting debug event type	Pseudocode
Software Access debug event	Pseudocode description of Software Access debug event
Exception Catch debug event	Pseudocode description of Exception Catch debug events
Halting Step debug events	Pseudocode description of Halting Step debug events
External Debug Request debug event	Pseudocode description of External Debug Request debug events
Reset Catch debug events	Pseudocode description of Reset Catch debug event
OS Unlock Catch debug event	Pseudocode description of OS Unlock Catch debug event

E3.2 Halting Step debug events

Halting Step is a debug resource that a debugger can use to make the PE step through code one instruction at a time. This section describes the Halting Step debug events. It is divided into the following sections:

- [Overview of a Halting Step debug event.](#)
- [The Halting Step state machine.](#)
- [Using Halting Step.](#)
- [Detailed Halting Step state machine behavior.](#)
- [Synchronization and the Halting Step state machine.](#)
- [Stepping T32 IT instructions.](#)
- [Disabling interrupts while stepping.](#)
- [Syndrome information on Halting Step.](#)
- [Pseudocode description of Halting Step debug events.](#)

The architecture describes the behavior as a simple Halting Step state machine. See [The Halting Step state machine.](#)

E3.2.1 Overview of a Halting Step debug event

The behavior of Halting Step is defined by a state machine, shown in [Figure E3-1](#). A Halting Step debug event executes a single instruction and then returns control to the debugger. When the debugger software wants to execute a Halting Step:

1. With the PE in Debug state, the debugger activates Halting Step.
2. The debugger signals the PE to exit Debug state and return to the instruction that is to be stepped.
3. The PE executes that single instruction.
4. The PE enters Debug state before executing the next instruction.

However, an exception might be generated during this process. That is, one of the following occurs:

- A synchronous exception generated by the instruction being stepped.
- An asynchronous exception taken before or during the instruction being stepped, to the extent permitted by the definition of a precise exception, without completing execution of the instruction being stepped.
- An asynchronous exception taken after completing execution of the instruction being stepped, before the PE enters Debug state.

See also [Definitions of synchronous and asynchronous exceptions.](#)

Halting Step has its own enable control bit, [EDECR.SS](#) and [EDES.R.SS](#).

Note

Because the Halting Step state machine states occur as a result of normal PE operation, the states can be described as both:

- PE states.
- Halting Step states.

E3.2.2 The Halting Step state machine

The state machine states are:

Inactive Halting Step is inactive. No Halting Step debug events can be generated, therefore execution is not affected by Halting Step. The PE is in this state whenever either of the following is true:

- Halting Step is disabled. That is, [EDECR.SS](#) is set to 0 and [EDES.R.SS](#) is set to 0.

- Halting is prohibited. See [Halting the PE on debug events](#). In this state, if `EDEC.R.SS` is set to 1, then a Halting Step debug event is pending.

In [Figure E3-1](#), this state is shown in red.

Active-not-pending

Halting Step is enabled and active. This is the state in which the PE steps an instruction. `EDEC.R.SS` == 1 and `EDES.R.SS` == 0. Software must not set `EDEC.R.SS` to 1 unless the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE, as described in [Changing the value of EDEC.R.SS when not in Debug state](#).

In [Figure E3-1](#), this state is shown in green.

Active-pending

Halting Step is enabled and active. The step has completed and the PE enters Debug state. `EDES.R.SS` == 1.

In [Figure E3-1](#), this state is shown in green.

Whenever Halting Step is enabled and active, whether the state machine is in the active-not-pending state or in the active-pending state depends on `EDES.R.SS`. [Table E3-3](#) shows this.

In the simple sequential execution of the program, the PE executes the Halting Step state machine as follows:

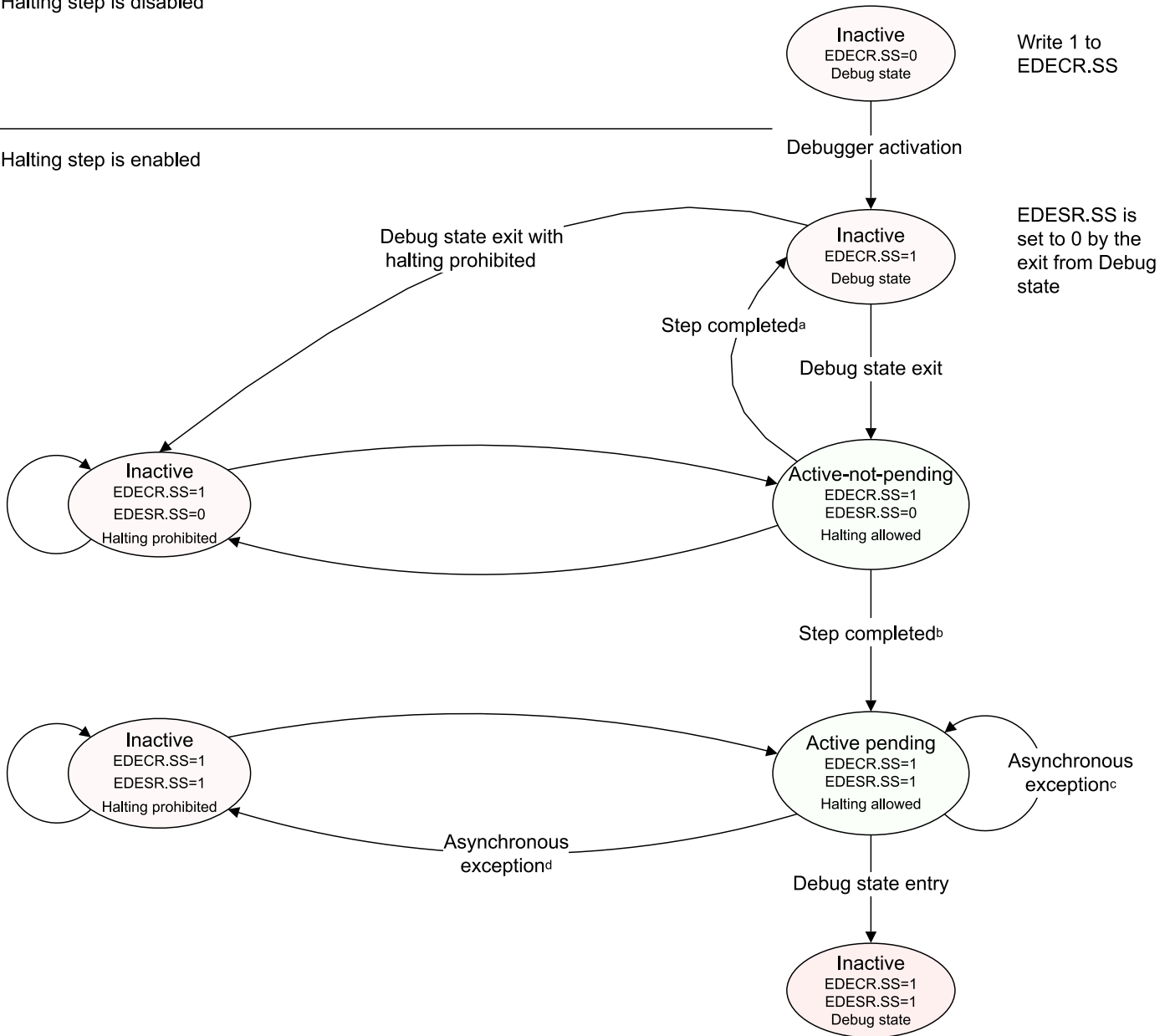
1. Initially, Halting Step is inactive.
2. After exiting Debug state, Halting Step is active-not-pending.
3. The PE executes an instruction and Halting Step is active-pending.
4. The pending Debug state entry is taken on the next instruction and the step is complete.

Exceptions and other changes to the PE context can interrupt this sequence.

[Figure E3-1](#) shows a Halting Step state machine.

Halting step is disabled

Halting step is enabled



- a. Step completed occurs when:
- A debug event, other than a Halting Step debug event, causes entry into Debug state.
- b. Step completed occurs when:
- An instruction is executed without taking an exception.
 - An exception is taken to a state where halting is allowed.
 - A reset.
- c. An asynchronous exception taken to a state where halting is allowed.
- d. An asynchronous exception taken to Secure state where halting is prohibited.

Figure E3-1 Halting Step state machine

Note

Figure E3-1 describes only state transitions to and from the inactive state by exit from Debug state, executing an exception return, or taking an exception. Other changes to the PE context, including writes to registers such as EDECR and DBGOSDLR and changes to the authentication interface can also cause changes to the Halting Step state machine. These can lead to UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior. See Synchronization and the Halting Step state machine.

The following bits control the state machine, as shown in Table E3-3:

- EDECR.SS. This is the Halting Step enable bit.
- EDESR.SS. This is the Halting Step debug event pending bit.

Table E3-3 shows the Halting Step state machine states. The letter X in a register column means that the relevant bit can be set to either zero or one.

Table E3-3 Halting Step state machine states

Halting	EDECR.SS	EDESR.SS	Halting Step state
Prohibited	X	X	Inactive (Halting Step debug even not pending)
Prohibited	X	1	Inactive (Halting Step debug event pending)
Allowed	0	0	Inactive
Allowed	1	0	Active-not-pending
Allowed	X	1	Active-pending

E3.2.3 Using Halting Step

To step a single instruction the PE must be in Debug state:

1. The debugger sets EDECR.SS to 1 to enable Halting step.
2. The debugger signals the PE to exit Debug state with DLR set to the address of the instruction being stepped. The PE clears EDESR.SS to 0 and the Halting Step state machine enters the active-not-pending state.
3. The PE executes the instruction being stepped.
If an exception is taken to a state where halting is prohibited, then EDESR.SS is always correct for the preferred return address of the exception.
4. The PE enters Debug state before executing the next instruction and the step is complete.

Note

- If FEAT_DoPD is not implemented, EDECR.SS value is in the Debug power domain, meaning that the state machine is maintained over a powerdown of the Core power domain.
- If FEAT_DoPD is implemented, the values of EDECR.SS and EDESR.SS are set to 0 on a Cold reset, and, if the PE was stepping an instruction, EDESR.SS is effectively UNKNOWN after a Warm reset. A debugger must use a Reset Catch debug event to step over a powerdown state.
- A debugger must only change the value of EDECR.SS when the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE as described in Changing the value of EDECR.SS when not in Debug state.

E3.2.4 Detailed Halting Step state machine behavior

The behavior of the Halting Step state machine is described in the following sections:

- [Entering the active-not-pending state.](#)
- [PE behavior in the active-not-pending state.](#)
- [Entering the active-pending state.](#)
- [PE behavior in the active-pending state.](#)
- [PE behavior in the inactive state when in Non-debug state.](#)
- [PE behavior in Debug state.](#)

E3.2.4.1 Entering the active-not-pending state

The PE enters the active-not-pending state:

- By exiting Debug state to a state where halting is allowed with `EDEC.R.SS == 1`.
- By an exception return from a state where halting is prohibited to a state where halting is allowed with `EDEC.R.SS == 1` and `EDES.R.SS == 0`.
- As described in [Synchronization and the Halting Step state machine](#).

E3.2.4.2 PE behavior in the active-not-pending state

When the PE is in the active-not-pending state it does one of the following:

- It executes one instruction and does one of the following:
 - Completes it without taking a synchronous exception.
 - Takes a synchronous exception generated by the instruction.
 - Generates a debug event that causes entry to Debug state.
- It takes an asynchronous exception without completing execution of the instruction.
- It takes an asynchronous debug event into Debug state.

E3.2.4.2.1 If no exception or debug event is generated

If no exception or debug event is generated the PE sets `EDES.R.SS` to 1. This means that the Halting Step state machine advances to the active-pending state.

E3.2.4.2.2 If an exception or debug event is generated

The PE sets `EDES.R.SS` according to all of the following:

If an exception or debug event is generated, the PE sets `EDES.R.SS` to 1 if the exception is an SMC exception or is taken to a state where halting is allowed.

Otherwise, `EDES.R.SS` is unchanged.

It is IMPLEMENTATION SPECIFIC whether `EDES.R.SS` is set to 1 or unchanged when an SError exception is taken to EL2 without completing execution of the instruction, and halting is prohibited at EL2.

If halting is prohibited after taking the exception or debug event, then the Halting Step state machine advances to the inactive state. Otherwise, the Halting Step state machine advances to the active-pending state.

Note

The underlying criteria for the value of `EDES.R.SS` on an exception are:

- Whether halting is allowed at the target of the exception. If halting is allowed, the PE must step into the exception. If halting is prohibited, the PE must step over the exception.
 - Whether the preferred return address of the exception is the instruction itself or the next instruction, if the PE steps over the exception.
-

shows the behavior of the active-not-pending state.

E3.2.4.3 Entering the active-pending state

The PE enters the active-pending state by one of the following:

- From the active-not-pending state by:
 - Executing an instruction without taking an exception.
 - Taking an exception so that the PE remains in a state where halting is allowed.
- An exception return from a state where halting is prohibited when `EDES.R.SS == 1`.
- A reset when the value of `EDEC.R.SS == 1`, regardless of the state the PE was in before the reset occurred.
- From the active-pending state by taking an asynchronous exception to a state where halting is allowed.
- Following the description in [Synchronization and the Halting Step state machine](#).

E3.2.4.4 PE behavior in the active-pending state

When the PE is in the active-pending state, it enters Debug state before executing an instruction.

The entry into Debug state has higher priority than all other types of synchronous debug event and synchronous exception. However, the architecture does not define the prioritization of this Debug state entry with respect to any unmasked pending asynchronous exception. If an asynchronous exception is prioritized over the entry to Debug state, then `EDES.R.SS` is unchanged.

For more information on the prioritization of debug events, see [Debug state entry and debug event prioritization](#).

E3.2.4.5 PE behavior in the inactive state when in Non-debug state

`EDES.R.SS` is not updated by the execution of an instruction or the taking of an exception when Halting Step is inactive. This means that `EDES.R.SS` is not changed by an exception handled in a state where halting is prohibited.

On return to a state where halting is allowed, the Halting Step state machine is restored either to the active-pending state or the active-not-pending state, depending on the value of `EDES.R.SS`. The return to a state where halting is allowed is normally by an exception return, which in some situations is a [Context Synchronization event](#).

See also [Synchronization and the Halting Step state machine](#).

E3.2.4.6 PE behavior in Debug state

Halting Step is inactive in Debug state because halting is prohibited, see [Halting allowed and halting prohibited](#).

Entry to Debug state does not change `EDES.R.SS`.

`EDES.R.SS` is cleared to 0 on exiting Debug state as the result of a restart request. If `EDEC.R.SS == 1`, Halting Step enters the active-not-pending state.

Note

This means that `EDES.R.SS` is never cleared to 0 by the execution of an instruction in Debug state, or by taking an exception when in Debug state as described in [PE behavior in the active-not-pending state](#), because the Halting Step state machine is not in the active-not-pending state. `EDES.R.SS` can be cleared by a write to `EDES.R`, see the register description.

However, if the PE exits Debug state as the result of a PE reset and `EDEC.R.SS == 1`, then Halting Step immediately enters the active-pending state, as `EDES.R.SS` is set to the value of `EDEC.R.SS`.

E3.2.5 Synchronization and the Halting Step state machine

The Halting Step state machine also changes state if:

- Halting becomes allowed or prohibited other than by exit from Debug state, an exception return, or taking an exception. This means that halting becomes allowed or prohibited because:
- A write to `EDEC.R` when the PE is in Non-debug state changes the value of `EDEC.R.SS`.

Note

Behavior is CONSTRAINED UNPREDICTABLE if the value of `EDEC.R.SS` is changed when the PE is in Non-debug state, see [Changing the value of EDEC.R.SS when not in Debug state](#).

- A write to `EDES.R` when the PE is in Non-debug state clears `EDES.R.SS` to 0.

These operations are guaranteed to take effect only after a [Context Synchronization event](#). If the instruction being stepped generates a [Context Synchronization event](#), then the PE might use the old or new state.

The PE must perform the required behavior of the new state before or immediately following the next [Context Synchronization event](#), but it is not required to do so immediately. This means that the PE can perform the required behavior of the old state before the next [Context Synchronization event](#). This is illustrated in [Example E3-1](#) and [Example E3-2](#).

Example E3-1 Synchronization requirements 1

`EDEC.R.SS` is set to 1 in Debug state, requesting the active-not-pending state on exit from Debug state. On exit from Debug state the PE immediately takes an exception to , where halting is prohibited. The PE does not step any instructions but executes the software at as normal. `EDES.R.SS` remains set to 0. If halting is subsequently allowed, the PE must perform the required behavior of the active-not-pending state before or immediately following the next [Context Synchronization event](#), but it is not required to do so immediately.

Example E3-2 Synchronization requirements 2

`EDEC.R.SS` is set to 1 in Debug state. On exit from Debug the PE executes an `MSR` instruction that sets `OSDLR_EL1.DLK` to 1 and `DoubleLockStatus()` becomes TRUE. This change requires a [Context Synchronization event](#) to guarantee its effect, meaning it is CONSTRAINED UNPREDICTABLE whether:

- Halting is allowed:
 - The PE enters Debug state on the next instruction.
- Halting is prohibited:
 - The PE does not enter Debug state.

The value in `EDES.R.SS` depends on whether halting was allowed or prohibited when the write to `OSDLR_EL1.DLK` completed, and so it might be 0 or 1. If a second `MSR` instruction clears `OSDLR_EL1.DLK` to 0, the PE must perform the required behavior of the state indicated by `EDES.R.SS` before or immediately following the next [Context Synchronization event](#), but it is not required to do so immediately.

See also [Synchronization and Halting debug events](#).

E3.2.5.1 Changing the value of EDEC.R.SS when not in Debug state

If software changes the value of `EDEC.R.SS` when the PE is not in Debug state then behavior is CONSTRAINED UNPREDICTABLE, and one or more of the following behaviors occurs:

- The value of `EDEC.R.SS` becomes UNKNOWN.
- The state of the Halting Step state machine becomes UNKNOWN.
- On a reset of the PE, the value of `EDEC.R.SS` and the state of the Halting Step state machine are UNKNOWN.

E3.2.6 Stepping T32 IT instructions

In an implementation that supports the ITD control, the architecture permits a combination of one T32 IT instruction and another 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

For the purpose of stepping an item, it is IMPLEMENTATION DEFINED whether:

- The PE considers such a pair of instructions to be one instruction.
- The PE considers such a pair of instructions be two instructions.

It is IMPLEMENTATION DEFINED whether this behavior depends on the value of the applicable ITD bit. For example:

- The debug logic might consider such a pair of instructions as one instruction, regardless of the state of the applicable ITD field.
- The debug logic might consider such a pair of instructions as two instructions, regardless of the state of the applicable ITD field.
- The debug logic might consider such a pair of instructions as one instruction when the value of the applicable ITD field is 1, and as two instructions when the value of the ITD field is 0.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

The ITD control fields are:

HSCTLR.ITD

Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD

Applies to execution at EL0 or EL1 when EL1 is using AArch32.

E3.2.7 Disabling interrupts while stepping

When using Halting Step, the sequence of entering Debug state, interacting with the debugger, and then exiting Debug state for each instruction reduces the rate at which the PE executes instructions. However, the rate at which certain interrupts, such as timer interrupts, are generated might be fixed by the system. This means it might be necessary to disable interrupts while using Halting Step by setting [EDSCR.INTdis](#), to allow the code being debugged to make forward progress.

E3.2.8 Syndrome information on Halting Step

On entering Debug state on a Halting Step debug event, the PE optionally records information about the debug event:

Halting Step, normal

An instruction other than a Load-Exclusive instruction was stepped.

Halting Step, exclusive

A Load-Exclusive instruction was stepped.

Halting Step, no syndrome

The syndrome data is not available.

If no syndrome is recorded, then [EDSCR.STATUS](#) is set to Halting Step, no syndrome.

If no instruction was stepped because Halting step entered the active-pending state from the inactive state without passing through the active-not-pending state, then no syndrome is recorded.

This happens when the active-pending state is entered for other reasons. See [Synchronization and the Halting Step state machine](#).

When a syndrome is recorded, [EDSCR.STATUS](#) is set to one of:

- Halting step, normal, if the stepped instruction was not a Load-Exclusive instruction.
- Halting step, exclusive, if the stepped instruction was a Load-Exclusive instruction.

Note

A Load-Exclusive instruction is any instruction that has a mnemonic starting with either LDREX or LDAEX

Note

Arm deprecates the use of the Halting Step, normal and Halting Step, exclusive syndrome values, and recommends software treats these values as Halting Step, no syndrome.

Table E3-4 summarizes the possible values that the PE can record in EDSCR.STATUS.

Table E3-4 Values that the PE can record in EDSCR.STATUS

Description	EDSCR.STATUS
Syndrome data is not available.	Halting Step, no syndrome.
Syndrome data is available. The instruction stepped was an instruction other than a Load-Exclusive instruction.	Syndrome data is available. Halting Step, normal.
Syndrome data is available. The instruction stepped was a Load-Exclusive instruction.	Syndrome data is available. Halting Step, exclusive.

E3.2.9 Pseudocode description of Halting Step debug events

There are two pseudocode functions for Halting Step debug events:

- `HSAdvance()`. This is called after an instruction has executed and any exception generated by the instruction is taken, except for the case of an SMC exception, in which case it is called before the exception is taken. `HSAdvance()` affects the next instruction.
- `CheckHaltingStep()`. This is called before the next instruction is executed. If a step is pending, it generates the debug event.

E3.3 Halt Instruction debug event

A Halt Instruction debug event is generated when `EDSCR.HDE == 1`, halting is allowed, and the PE executes the Halt instruction, `HLT`.

The pseudocode for Halt Instruction debug events is described in `HLT` for T32 and A32.

`HLT` never generates a debug exception. It is treated as `UNDEFINED` if `EDSCR.HDE == 0`, or if halting is prohibited.

Note

A debugger can replace a program instruction with a Halt instruction to generate a Halt Instruction debug event. Debuggers that use the `HLT` instruction must be aware of the rules for concurrent modification of executable code, `CMODX`. The rules for concurrent modification and execution of instructions do not allow one thread of execution or an external debugger to replace an instruction with an `HLT` instruction when these same instructions are being executed by a different thread of execution. See Concurrent modification and execution of instructions.

The T32 `HLT` instruction is unconditionally executed inside an `IT` block, even when it is treated as undefined. The A32 `HLT` instruction is `CONSTRAINED UNPREDICTABLE` if the Condition code field is not `0b1110`, with the set of behaviors the same as for `BKPT`. See [Architectural Constraints on UNPREDICTABLE Behaviors](#).

Note

The `HLT` instruction is part of the external debug solution from the introduction of Armv8-A. As such, the presence of the `HLT` instruction is not indicated in the ID registers. In particular, the AArch32 System register `ID_ISAR0`. Debug does not indicate the presence of the `HLT` instruction.

E3.3.1 HLT instructions as the first instruction in a T32 IT block

In an implementation that supports the ITD control, the architecture permits a combination of one T32 `IT` instruction and certain other 16-bit T32 instruction to be treated as a single 32-bit instruction when the value of the ITD field that applies to the current Exception level is 1.

The T32 `HLT` instruction cannot be combined with an `IT` instruction in this way. In an implementation that supports the ITD control, if the first instruction in an IT block is an `HLT` instruction, then the behavior of the instruction depends on the value of the applicable ITD field:

- If the value of the ITD field is 1, then the combination is treated as undefined and an Undefined Instruction exception is generated either by the `IT` instruction or by the `HLT` instruction.
- If the value of the ITD field is 0, then the `HLT` instruction unconditionally executed.

An implementation that does not support the ITD control behaves as if the value of the ITD field is 0.

To set an Halt Instruction debug event on the first instruction of an IT block, debuggers must replace the `IT` instruction with an `HLT` instruction to ensure consistent behavior.

The ITD control fields are:

HSCTLR.ITD

Applies to execution at EL2 when EL2 is using AArch32.

SCTLR.ITD

Applies to execution at EL0 or EL1 when EL1 is using AArch32.

Note

An `HLT` instruction is always unconditional, even within an IT block.

E3.4 Exception Catch debug event

Exception Catch debug events:

- Are generated when the corresponding bit in the Exception Catch Control Register, [EDECCR](#), is set to 1 on all entries to a given Exception level. This means:
 - Exceptions taken to the Exception level.
 - Exception returns to the Exception level.
 - It is IMPLEMENTATION DEFINED whether a reset into an Exception level generates an Exception Catch debug event.
- Might be taken synchronously, after the exception or reset entry or the exception return has been processed by the PE.
- Ignore the Execution state of the target Exception level.
- Might be ignored if halting is prohibited.

The [EDECCR](#) register contains fields that control when Exception Catch debug events are generated.

See Controlling Exception Catch debug events when FEAT_Debugv8p2 is implemented and [Controlling Exception Catch debug events when FEAT_Debugv8p2 is not implemented](#).

For exception returns, the final Exception level of the exception return determines whether an Exception Catch debug event is generated. On an illegal exception return, an Exception Catch debug event is generated only if EDECCR is programmed to generate an Exception Catch debug event for an exception return to the current Exception level.

E3.4.1 Prioritization of Exception Catch debug events

The following rules define the prioritization of Exception Catch debug events:

- It is IMPLEMENTATION DEFINED whether Exception Catch debug events are higher or lower priority than Software Step exceptions and [Halting Step debug events](#).
- Exception Catch debug events are higher priority than all synchronous exceptions other than Software Step exceptions and debug events other than [Halting Step debug events](#).
- Exception Catch debug events are lower priority than [Reset Catch debug events](#).
- Exception Catch debug events are higher priority than pending asynchronous exceptions. Otherwise, the prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event, is IMPLEMENTATION DEFINED.

E3.4.2 Generating Exception Catch debug events

When an Exception Catch debug event is generated after exception entry and halting is allowed at the target of the exception, the PE halts and enters Debug state:

- The PE halts and enters Debug state before the first instruction at the handler is executed, after the exception entry has updated the program counter, PSTATE, and syndrome registers for the exception.
- The PE does not fetch instructions from the vector address before entering Debug state, if address translation is disabled in the translation regime at the target Exception level.
- A second unmasked asynchronous exception can be taken before the PE enters Debug state. If this second exception does not generate an Exception Catch debug event, the exception handler executed at the higher Exception level later returns to the trapped Exception level, causing the Exception Catch debug event to be generated again.

When an Exception Catch debug event is generated on exception return and halting is allowed at the target of the exception return:

- There is no prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event generated on an exception return.
- The PE halts and enters Debug state after the exception return has updated the program counter and PSTATE, and before the execution of the first instruction at the return address is completed.

Otherwise, when the PE is executing code at a given Exception level, and the corresponding [EDECCR](#) bit is 1, it is CONstrained UNpredictable whether an Exception Catch debug event is generated.

Examples of this are:

- If the debugger writes to [EDECCR](#) so that the current Exception level is trapped.
- If the OS restore code writes to [DBGOSECCR](#) so that the current Exception level is trapped.
- If the code executing in AArch32 state changes the Exception level other than by an exception return, and the target Exception level is trapped. See [State and mode changes without explicit Context Synchronization events](#).

See also [Debug state entry and debug event prioritization](#).

Note

It is possible to generate Exception Catch debug events:

- As a trap on all instruction fetches from the trapped Exception level as part of an instruction fetch.
- On entry to the Exception level, as described in [Detailed Halting Step state machine behavior](#).

This is similar to the implementation options allowed for Vector Catch debug events. The architecture does not require that the event is generated following an [ISB](#) operation executed at the Exception level.

E3.4.3 Controlling Exception Catch debug events

All Exception Catch debug events are enabled by a combination of the fields NSE and SE in [EDECCR](#), as shown in [Table E3-5](#).

Table E3-5 Summary of Exception Catch debug event control

(N)SE<n>	Behavior on exception taken to ELn, return to ELn, and if resets are Exception Catch debug events, reset into ELn
0	No action.
1	Halt if allowed.

E3.4.4 Examples of Exception Catch debug events

If [EDECCR](#) == 0x0020, meaning that the Exception Catch debug event is enabled for EL1, then the following exceptions generate Exception Catch debug events:

- An exception taken from EL0 to EL1.
- An exception return from EL2 to EL1.

For example, on taking a Data Abort exception from EL0 to EL1:

- LR and [SPSR_abt](#) are written with the preferred return address and PE state for a return to EL0.
- ESR and FAR are written with the syndrome information for the exception.
- [DLR](#) is set to VBAR + 0x400, the synchronous exception vector.
- [DSPSR](#) is written with the PE state for an exit to EL1.

The following do not generate Exception Catch debug events:

- An exception taken from EL0 to EL2.
- An exception return from EL2 to EL0.
- An exception taken from EL0 to EL1.

E3.4.5 Pseudocode description of Exception Catch debug events

The pseudocode functions `CheckExceptionCatch()` and `CheckPendingExceptionCatch()` are described in [R-profile Architecture Pseudocode](#).

E3.5 External Debug Request debug event

External Debug Request debug events are asynchronous debug events.

An External Debug Request debug event can be generated by any of the following:

- When signaled by the embedded cross-trigger. See [The Embedded Cross-Trigger Interface](#).
- The Performance Monitoring Unit. See [PMU Overflow external debug request](#).
- An implementation might also support IMPLEMENTATION DEFINED ways of generating an External Debug Request debug event.

E3.5.1 Synchronization and External Debug Request debug events

For all Context synchronization events, if an External Debug Request debug event is asserted before the [Context Synchronization event](#), and the External Debug Request debug event remains asserted and halting is allowed after the Context synchronization event, then the debug event is taken and the PE enters Debug state before the first instruction following the Context synchronization event completes its execution.

An External Debug Request debug event that is being asserted when the PE comes out of reset is taken, and the PE enters Debug state before the first instruction after the reset completes its execution, provided that halting is allowed when the PE exits reset state.

If the first instruction after the [Context Synchronization event](#) or after coming out of reset generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken.

Otherwise, when all of the following are true, External Debug Request debug events must be taken in finite time, without requiring the synchronization of any necessary change to the external authentication interface:

- Halting is allowed.
- The PE is not in a low-power state that is not required to exit on an external debug request. See [Core power domain power states](#).

External debug request is a wakeup event for WFI, WFE, or WFET instructions. See Mechanisms for entering a low-power state.

Note

These rules are based on the rules that apply when taking asynchronous exceptions. See Asynchronous exception types.

If an unmasked External Debug Request debug event was pending but is changed to not pending before it is taken, then the architecture permits the External Debug Request debug event to be taken, but does not require this to happen. If the External Debug Request debug event is taken then it must be taken before the first [Context Synchronization event](#) after the External Debug Request debug event was changed to not pending.

[Example E3-3](#) shows an example of the synchronization requirements.

Example E3-3 Synchronization requirements

Secure software locks up in a tight loop, so it executes indefinitely without any synchronization operations. An External debug request must be able to break the PE out of that loop. This is a requirement even if **DBGEN** or **SPIDEN** or both are LOW on entry to the loop, meaning that halting is prohibited, and are only asserted HIGH later.

E3.5.2 PMU Overflow external debug request

When the PE halts and enters Debug state because of a PMU Overflow external debug request:

- DLR_EL0 is set to the address of the first instruction that must be executed on exit from Debug state. DSPSR_EL0 is correct for this instruction.

- [EDSCR.STATUS](#) is set to 0b010011, external debug request.

E3.5.3 Pseudocode description of External Debug Request debug events

The [ExternalDebugRequest \(\)](#) function is described in [R-profile Architecture Pseudocode](#).

E3.6 OS Unlock Catch debug event

An OS Unlock Catch debug event is generated when enabled and the state of the OS Lock changes from locked to unlocked. When FEAT_DoPD is implemented, CTIDEVCTL.OSUCE enables an OS Unlock Catch debug event, otherwise EDECR.OSUCE enables an OS Unlock Catch debug event.

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled, and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. However, this is an indirect write to EDESR.OSUC, meaning the OS Unlock Catch debug event is not guaranteed to be taken before a subsequent Context synchronization event. If the PE enters Debug state or the OS Unlock Catch debug event is disabled before EDESR.OSUC becomes set to 1, then EDESR.OSUC might not be set.

OS Unlock Catch debug events are not generated if the OS Lock is unlocked when the PE is in Debug state. See also [Synchronization and Halting debug events](#).

EDESR.OSUC is cleared to 0 on a Warm reset and on exiting Debug state.

E3.6.1 Using the OS Unlock Catch debug event

When the Core power domain is completely off or in a low-power state, a debugger is permitted to access a debug register that is implemented in the External debug power domain. However, if a debugger attempts to access a debug register that is implemented in the Core power domain when the Core power domain registers cannot be accessed, and that access returns an error, the debugger must retry the access.

Regularly powering down the Core power domain can result in unreliable debugger behavior.

The debugger can program a Reset Catch debug event to halt the PE when it has powered up, and can program the debug registers from Debug state. However, if the PE boot software restores the debug registers, as described in [Debug OS Save and Restore sequences](#), then newly written values are overwritten by the restore sequence.

The debugger can program an OS Unlock Catch debug event to halt the PE after the restore sequence has completed, and program the debug registers from Debug state.

E3.6.2 Pseudocode description of OS Unlock Catch debug event

The `CheckOSUnlockCatch()` function is called when the OS Lock is unlocked.

The `CheckPendingOSUnlockCatch()` function is called before an instruction is executed. If an OS Unlock Catch is pending, it generates the debug event.

E3.7 Reset Catch debug events

A Reset Catch debug event is generated when enabled, and the PE exits reset state. When the Reset Catch debug event is generated, it is recorded by setting [EDES.RC](#) to 1. When FEAT_DoPD is implemented, CTIDEVCTL.RCE enables a Reset Catch debug event, otherwise [EDECR.RCE](#) enables a Reset Catch debug event.

If halting is allowed when the event is generated, the Reset Catch debug event is taken immediately and synchronously. On entering Debug state, [DLR](#) has the address of the reset vector. The PE must not fetch any instructions from memory.

Otherwise, the Reset Catch debug event is pended and taken when halting is allowed. See [Synchronization and Halting debug events](#) for more information.

This means that [EDES.RC](#) is set to the value of [EDECR.RCE](#) or CTIDEVCTL.RCE on a Warm reset. [EDES.RC](#) is cleared to 0 on exiting Debug state.

E3.7.1 Pseudocode description of Reset Catch debug event

The [CheckResetCatch\(\)](#) function is called after reset before executing any instruction.

The [CheckPendingResetCatch\(\)](#) function is called before an instruction is executed. If a Reset Catch is pending, it generates the Reset Catch debug event.

E3.8 Software Access debug event

When the value of [EDSCR.TDA](#) is 1, software access to the following debug System registers generate a Software Access debug event:

- The Breakpoint Value Registers, [DBGBVR<n>](#).
- The Breakpoint Control Registers, [DBGBCR<n>](#).
- The Watchpoint Value Registers, [DBGWVR<n>](#).
- The Watchpoint Control Registers, [DBGWCR<n>](#).

However, [EDSCR.TDA](#) is ignored if any of the following applies:

- The value of [DBGOSLSR.OSLK](#) == 1, meaning that the OS Lock is locked.
- Halting is prohibited. See [Halting allowed and halting prohibited](#).
- The register access generates a higher priority synchronous exception or debug event.

Note

- The only accesses to the specified registers that generate a Software Access debug event are:
 - Accesses to System registers in the (`coproc == 0b1110`) encoding space in AArch32 state.
 - Accesses by a PE using the external debug interface never generate a Software Access debug event.
-

E3.8.1 Pseudocode description of Software Access debug event

The [CheckSoftwareAccessToDebugRegisters\(\)](#) function is described in .

E3.9 Synchronization and Halting debug events

The behavior of external debug depends on:

- Indirect reads of:
 - External debug registers.
 - System registers, including system debug registers.
 - Special-purpose registers.
- The state of the external authentication interface.

For some registers, all read and write accesses that update the register occur in program order, without any additional synchronization, but others require an explicit [Context Synchronization event](#). For more information on the synchronization of register updates, see:

- [Synchronization of changes to the external debug registers.](#)
- [State and mode changes without explicit Context Synchronization events.](#)

Changes to the external authentication interface do not require explicit synchronization to affect External Debug Request debug events. See [Synchronization and External Debug Request debug events](#).

For changes that require explicit synchronization, it is **CONSTRAINED UNPREDICTABLE** whether instructions between the change and the [Context Synchronization event](#) observe the old state or the new state.

This means that any change to these registers or the external authentication interface requires explicit synchronization by a [Context Synchronization event](#) before the change takes effect. This ensures that for instructions appearing in program order after the change, the change affects the following:

- The generation and behavior of Breakpoint and Watchpoint debug events. See [Synchronization and debug exceptions](#).
- The generation of all Halting debug events by instructions.
- Taking a pending Halting debug event or other asynchronous debug event. See:
 - [Pending Halting debug events.](#)
 - [Synchronization and External Debug Request debug events.](#)
- The behavior of the Halting Step state machine. See [Synchronization and the Halting Step state machine](#).

E3.9.1 Pending Halting debug events

A Halting debug event might be pending:

- If Halting Step of an instruction sets [EDES.RC](#) to 1, and halting is prohibited following the step, then the Halting Step state machine is inactive but a Halting Step debug event is pending.
- If a Reset Catch debug event sets [EDES.RC](#) to 1, and halting is prohibited following reset, then a Reset Catch debug event is pending.
- If an OS Unlock Catch debug event sets [EDES.OSUC](#) to 1, then an OS Unlock Catch debug event is pending.

Pending Halting debug events are taken asynchronously when halting is allowed.

Pending Halting debug events are discarded by a Cold reset. The debugger can also force a pending event to be dropped by writing to [EDES](#).

Any Halting debug event that is observed as pending in the [EDES](#) before a [Context Synchronization event](#) is taken and the PE enters Debug state before the first instruction following the [Context Synchronization event](#) completes its execution. This is possible only if halting is allowed after completion of the [Context Synchronization event](#).

Chapter E4

The Debug Communication Channel and Instruction Transfer Register

This chapter describes communication between a debugger and the implemented debug logic, using the *Debug Communications Channel* (DCC) and the *Instruction Transfer Register* (ITR), and associated control flags. It contains the following sections:

- [Introduction](#)
- [DCC and ITR registers](#)
- [DCC and ITR access modes](#)
- [Flow control of the DCC and ITR registers](#)
- [Synchronization of DCC and ITR accesses](#)
- [Interrupt-driven use of the DCC](#)
- [Pseudocode description of the operation of the DCC and ITR registers](#)

E4.1 Introduction

The *Debug Communications Channel*, DCC, is a channel for passing data between the PE and an external agent, such as a debugger. The DCC provides a communications channel between:

- An external debugger, described as the *debug host*.
- The debug implementation on the PE, described as the *debug target*.

The DCC can be used:

- As a 32-bit full-duplex channel.
- As a 64-bit half-duplex channel.

The DCC is an essential part of Debug state operation and can also be used in Non-debug state.

The *Instruction Transfer Register*, ITR, passes instructions to the PE to execute in Debug state.

The PE includes flow-control mechanisms for both the DCC and ITR.

E4.2 DCC and ITR registers

The DCC comprises *data transfer registers*, the DTRs, and associated flow-control flags. The data transfer registers are DTRRX and DTRTX.

The ITR comprises a single register, [EDITR](#), and associated flow-control flags.

In AArch32 state, software can access the data transfer registers only as:

- A receive and transmit pair, for 32-bit full duplex operation:
 - The write-only [DBGDTRTXint](#) register to transmit data.
 - The read-only [DBGDTRRXint](#) register to receive data.
- The read/write [DBGDTRTXext](#) and [DBGDTRRXext](#) registers for save and restore.

The data transfer registers are also accessible by the external debug interface as a pair of 32-bit registers, [DBGDTRRX_EL0](#) and [DBGDTRTX_EL0](#). Both registers are read/write, allowing both 32-bit full-duplex and 64-bit half-duplex operation.

The DCC flow-control flags are [EDSCR](#). {RXfull, TXfull, RXO, TXU}:

- The RXfull and TXfull ready flags are used for flow-control and are visible to software in the Debug System registers in [DBGDSCRint](#).
- The RX overrun flag, RXO, and the TX underrun flag, TXU, report flow-control errors.
- The flow-control flags are also accessible by software as simple read/write bits for saving and restoring over a powerdown when the OS Lock is locked in [DBGDSCRExt](#).
- The flow-control flags are accessible from the external debug interface in [EDSCR](#).

[Figure E4-1](#) shows the System register and external debug interface views of the [EDSCR](#) and DTR registers. These figures do not include the save and restore views.

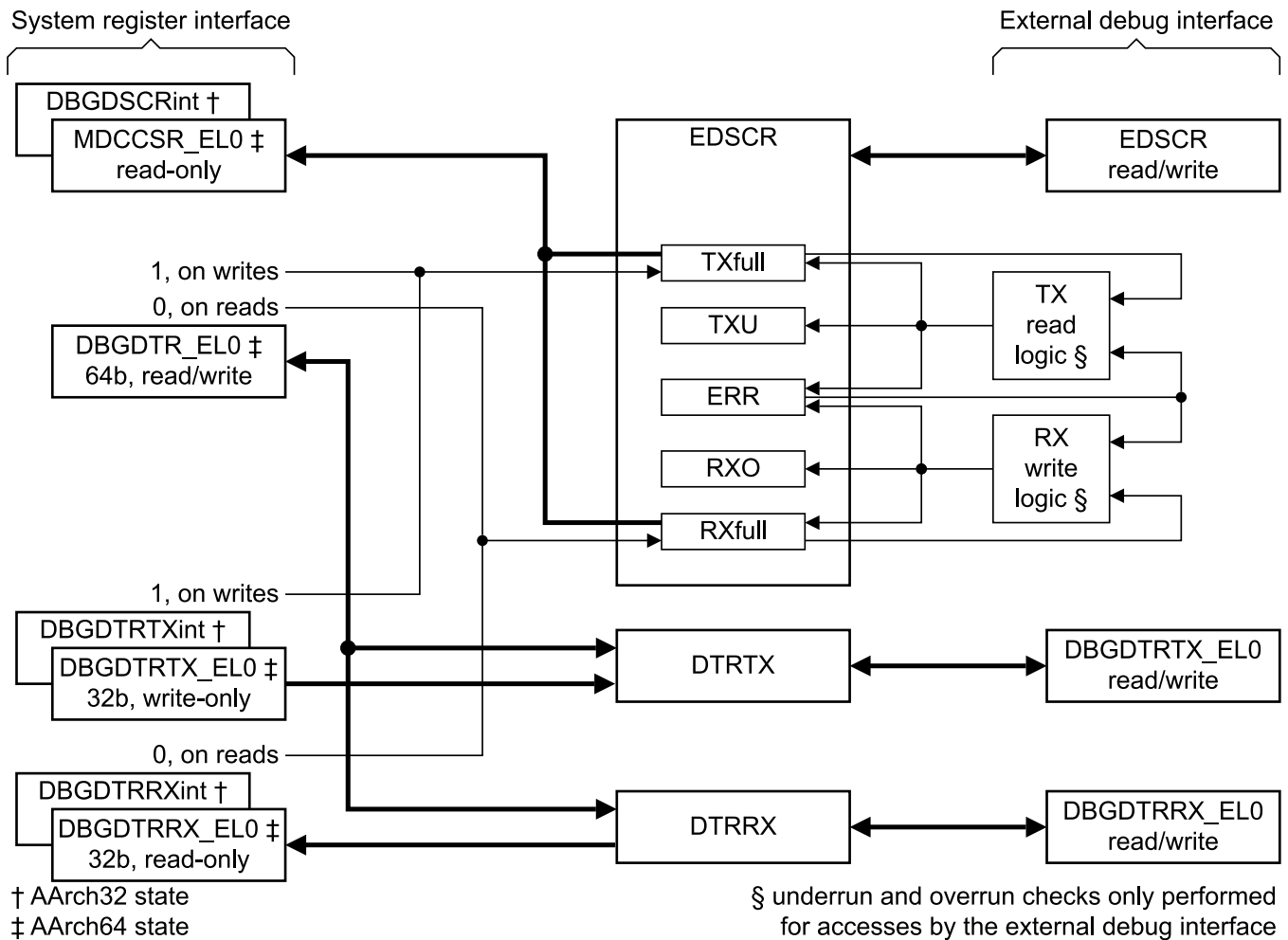


Figure E4-1 System register and external debug interface views of EDSCR and DTR registers, Normal access mode

EDITR and the ITR flow-control flags, **EDSCR**.{ITE, ITO} are accessible only by the external debug interface:

- The **EDITR** specifies an instruction to execute in Debug state.
- The ITR empty flag, ITE, is used for flow-control.
- The ITR overrun flag, ITO, reports flow-control errors.

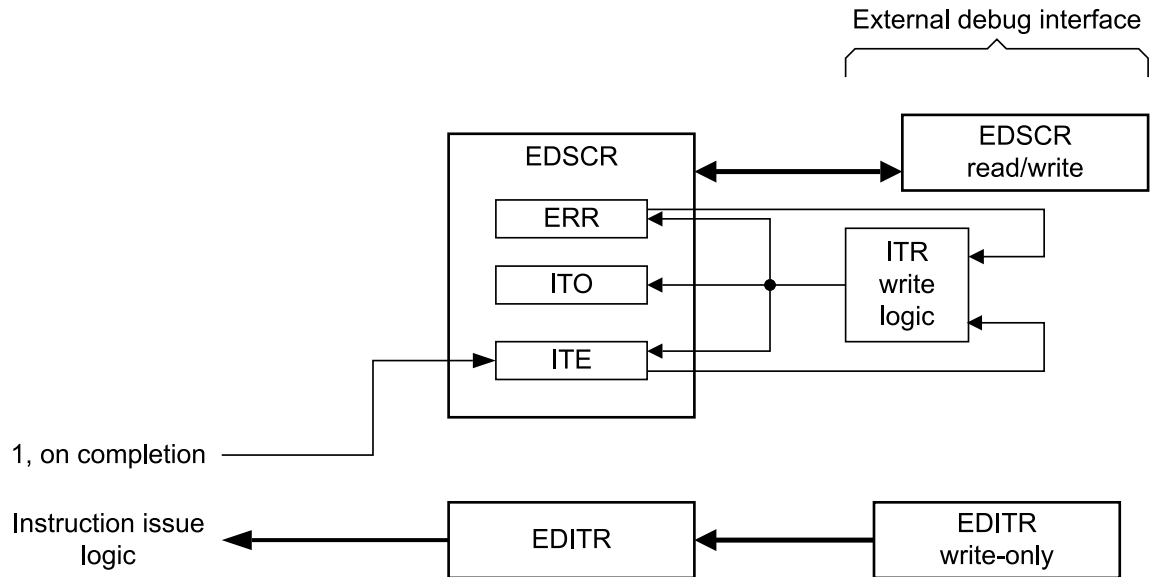


Figure E4-2 External debug interface views of EDSCR and EDITR registers, Normal access mode

The sticky overflow flag, [EDSCR.ERR](#), is used by both the DCC and ITR to report flow-control errors.

To save and restore the DCC registers for an external debugger over powerdown, software uses:

- The [DBGDSCRext](#), [DBGDTRTXext](#), and [DBGDTRRXext](#) registers in AArch32 state.

Note

There is no save and restore mechanism for the ITR registers as the ITR is used only in Debug state.

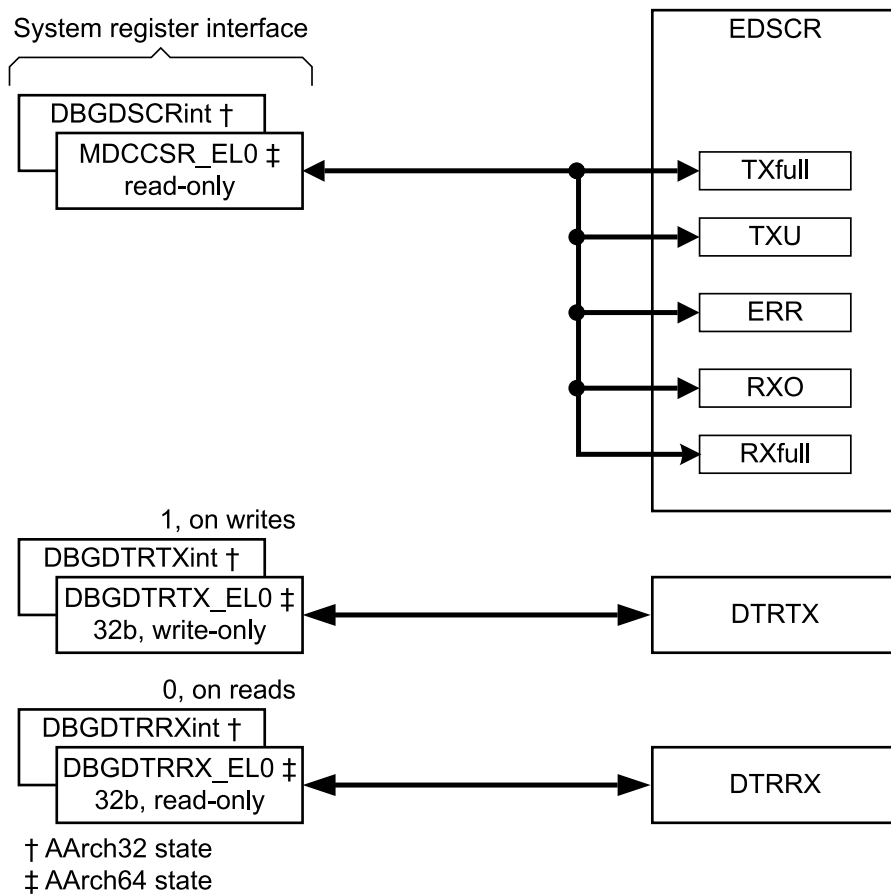


Figure E4-3 System register views of EDSCR and DTR registers for save and restore

E4.3 DCC and ITR access modes

The DCC and ITR support two access modes:

DCC and ITR access mode, links to description	Applies when:
Normal access mode	EDSCR.MA == 0 or the PE is in Non-debug state
Memory access mode	EDSCR.MA == 1 and the PE is in Debug state

E4.3.1 Normal access mode

The Normal access mode allows use of the DCC as a communications channel between target and host. It also allows the use of the ITR for issuing instructions to the PE in Debug state.

In Normal access mode, if there is no overrun or underrun, the following occurs:

For accesses by software:

- Direct writes to [DBGDTRTXint](#) update the value in DTRTX and indirectly write 1 to TXfull.
- Direct reads from [DBGDTRRXint](#) return the value in DTRRX and indirectly write 0 to RXfull.

Software reads TXfull and RXfull using [DBGDSCRint](#).

For accesses by the external debug interface:

- Writes to [EDITR](#) trigger the instruction to be executed if the PE is in Debug state:
 - If the PE is in AArch32 state, this is a T32 instruction. The T32 instruction is a pair of halfwords where the first halfword is taken from the lower 16-bits, and the second halfword is taken from the upper 16-bits.
- Reads of [DBGDTRTX_ELO](#) return the value in DTRTX and indirectly write 0 to TXfull.
- Writes to [DBGDTRTX_ELO](#) update the value in DTRTX and do not change TXfull.
- Reads of [DBGDTRRX_ELO](#) return the value in DTRRX and do not change RXfull.
- Writes to [DBGDTRRX_ELO](#) update the value in DTRRX and indirectly write 1 to RXfull. TXfull and RXfull are visible to the external debug interface in [EDSCR](#).

The PE detects overrun and underrun by the external debug interface, and records errors in [EDSCR](#). {TXU, RXO, ITO, ERR}. See [Flow control of the DCC and ITR registers](#).

See also [Synchronization of DCC and ITR accesses](#).

E4.3.2 Memory access mode

When the PE is in Debug state, *Memory access mode* can be selected to accelerate word-aligned block reads or writes of memory by an external debugger. Memory access mode can be enabled only in Debug state, and no instructions can be issued directly by the debugger when in Memory access mode.

If there is no overrun or underrun when in Memory access mode, an access by the external debug interface results in the following:

- External reads from [DBGDTRTX_ELO](#) cause:
 1. The existing value in DTRTX to be returned. This clears [EDSCR.TXfull](#) to 0.
 2. The equivalent of `LDR W1, [X0], #4`, if in AArch64 state, or `LDR R1, [R0], #4`, if in AArch32 state, to be executed.
 3. The equivalent of the `MSR DBGDTRTX_ELO, X1` instruction, if in AArch64 state, or the `MCR p14, 0, R1, c0, c5, 0` instruction, if in AArch32 state, to be executed.

4. [EDSCR](#).{TXfull, ITE} to be set to {1, 1}, and X1 or R1 to be set to an UNKNOWN value.
- External writes to [DBGDTRRX_EL0](#) cause:
 1. The value in DTRRX to be updated. This sets [EDSCR](#).RXfull to 1.
 2. The equivalent of the instruction `MRS X1, DBGDTRRX_EL0`, if in AArch64 state, or `MRC p14, 0, R1, c0, c5, 0` if in AArch32 state, to be executed.
 3. The equivalent of the instruction `STR W1, [X0], #4`, if in AArch64 state, or `STR R1, [R0], #4`, if in AArch32 state, to be executed.
 4. [EDSCR](#).{RXfull, ITE} to be set to {0, 1}, and X1 or R1 to be set to an UNKNOWN value.
- External reads from [DBGDTRRX_EL0](#) return the last value written to DTRRX.
- External writes to [EDITR](#) generate an overrun error.

During these accesses, [EDSCR](#).{TXfull, RXfull, ITE} are used for flow control.

Note

An overrun or underrun might result in [EDSCR](#).ERR being set to 1 asynchronously to the sequence of operations that are outlined in this section. As this is timing-dependent, it is UNPREDICTABLE when the [EDSCR](#).ERR flag affects the instructions and therefore whether neither instruction, only the first instruction, or both instructions are executed. If the second instruction is executed, then the first instruction must have been executed. However, in each case X1 or R1 is set to an UNKNOWN value. This means that:

- In both cases, if the memory access instruction is not executed, then the base register X0 or R0 is not updated, meaning the debugger can determine the last accessed location.
- In the list describing External reads from [DBGDTRTX_EL0](#), DTRTX and [EDSCR](#).TXfull get set to UNKNOWN values. If the load was executed, then the value that was read by the PE is lost. This means the operation might need to be repeated by the debugger, and it is not advisable to use Memory access mode to read from read-sensitive locations using the underrun and overrun detection for flow control.
- In the list describing External writes to [DBGDTRRX_EL0](#), [EDSCR](#).RXfull is set to an UNKNOWN value.

A Data Abort from the memory access can also set [EDSCR](#).ERR to 1. See [Data Aborts in Memory access mode](#).

The architecture does not require precisely when these flags are set or cleared by the sequence of operations outlined in this section.

The architecture does require that the flags are set as at step 4 when the PE is ready to accept a further read or write without causing an overrun error or an underrun error.

The process outlined in this section represents a simple sequential execution model of Memory access mode. An implementation is free to pipeline, buffer, and reorder instructions and transactions, as long as the following remain true:

- Data items are transferred into and out of the DTR in order and without loss of data, other than as a result of an overrun or an underrun.
- Data Aborts occur in order.
- The constraints of the memory type are met.
- In the list describing [External reads from](#):
 - The MSR equivalent operation at step 3 of the sequence reads the value loaded by step 2.
 - If the list is performed in a loop, for all but the first iteration of this list, the value read by step 1 returns the values written by the MSR equivalent operation at the previous iteration of step 3.
- In the list describing [External writes to](#):
 - The MRS equivalent operation at step 2 of the sequence returns the value written at step 1.
 - The STR equivalent at step 3 of the sequence writes the value read at step 2.
- If the PE cannot accept a read or write, as applicable, during the sequence, then the flags are updated to indicate an overrun or underrun.

See [Flow control of the DCC and ITR registers](#) for more information on overrun and underrun.

E4.3.2.1 Ordering, access sizes and effect on Exclusives monitors

For the purposes of memory ordering, access sizes, and effect on the Exclusives monitor, accesses in Memory access mode are consistent with load/store word instructions executed by the PE.

The simple sequential access model of Memory access mode, as stated in [Memory access mode](#), must also be ordered with respect to instructions executed as a result of External writes to [EDITR](#) in Normal mode both before and after accesses to the DTR registers in Memory access mode.

E4.3.2.2 Data Aborts in Memory access mode

If a memory access generates a Data Abort, then:

- The Data Abort exception is taken. See [Exceptions in Debug state](#):
 - This means [EDSCR.ERR](#) is set to 1, see [Cumulative error flag](#).
 - If the Data Abort occurs on stage 2 of an address translation, then the values returned in the ISV field and in bits[23:14] of the ISS are UNKNOWN.
If EL2 is using AArch32 and this Data Abort is taken to Hyp mode, the ISS is returned by [HSR. ISS encoding for exception from a Data Abort](#) describes the usual encoding of this ISS.
- Register R0 retains the address that generated the abort.
- Register R1 is set to an UNKNOWN value.
- [EDSCR.TXfull](#), for a load, or [EDSCR.RXfull](#), for a store, is set to an UNKNOWN value.
- DTRTX, for a load, or DTRRX, for a store, is set to an UNKNOWN value.
- [EDSCR.ITE](#) is set to 1.

E4.3.2.3 Illegal Execution state exception

If PSTATE.IL is set to 1 when [EDSCR.MA](#) = 1, then on an external write access to [DBGDTRRX_EL0](#) or an external read from [DBGDTRTX_EL0](#), it is CONSTRAINED UNPREDICTABLE whether the PE:

- Takes an Illegal Execution state exception without performing any operations. In this case:
 - [EDSCR.ERR](#) is set to 1, see [Cumulative error flag](#).
 - Register R0 is unchanged.
 - Register R1 is set to an UNKNOWN value.
 - [EDSCR.TXfull](#) or [EDSCR.RXfull](#), as applicable, is set to an UNKNOWN value.
 - DTRTX or DTRRX, as applicable, is set an UNKNOWN value.
 - [EDSCR.ITE](#) is set to 1.
 See also [Exceptions in Debug state](#).
- Ignores PSTATE.IL.

Note

The typical usage model for Memory access mode involves executing instructions in Normal access mode to set up X0 before setting [EDSCR.MA](#) to 1. These instructions generate an Illegal state exception if PSTATE.IL is set to 1.

E4.3.2.4 Alignment constraints

If the address in R0 is not aligned to a multiple of four, the behavior is as follows:

- For each external DTR access a CONSTRAINED UNPREDICTABLE choice of:
 1. The PE makes an unaligned memory access to R0. If alignment checking is enabled for the memory access, this generates an Alignment fault.
 2. The PE makes a memory access to $\text{Align}(R[0], 4)$.
 3. The PE generates an Alignment fault, regardless of whether alignment checking is enabled.
 4. The PE does nothing.
- Following each memory access, if there is no Data Abort, R0 is updated with an UNKNOWN value.
- For external writes to [DBGDTRRX_EL0](#), if the PE writes to memory, an UNKNOWN value is written.
- For external reads of [DBGDTRTX_EL0](#) an UNKNOWN value is returned.
- The RXfull and TXfull flags are left in an UNKNOWN state, meaning that a [DBGDTRTX_EL0](#) read can trigger a TX underrun, and a [DBGDTRTX_EL0](#) write can trigger an RX overrun.

E4.3.3 Memory-mapped accesses to the DCC and ITR

Writes to the flags in [EDSCR](#) by external debug interface accesses to the DCC and the ITR registers are indirect writes, because they are a side-effect of the access. The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see [Register access permissions for memory-mapped accesses](#).

E4.4 Flow control of the DCC and ITR registers

- Ready flags.
- Buffering writes to EDITR.
- Overrun and underrun flags.
- Cumulative error flag.

E4.4.1 Ready flags

In Normal access mode:

- For the DTR registers there are two ready flags:
 - `EDSCR.RXfull` == 1 indicates that `DBGDTRRX_EL0` contains a valid value that has been written by the external debugger and not yet read by software running on the target.
 - `EDSCR.TXfull` == 1 indicates that `DBGDTRTX_EL0` contains a valid value that has been written by software running on the target and not yet read by an external debugger.
- For the ITR register there is a single ready flag:
 - `EDSCR.ITE` == 1 indicates that the PE is ready to accept an instruction to the ITR.

Note

The architecture permits a PE to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, as long as those instructions are discarded if `EDSCR.ERR` is 1, either by an underrun or overrun or by any of the other error conditions described in this architecture, such as an instruction generating an abort.

In Memory access mode:

- `EDSCR.{RXfull, ITE}` == {0, 1} indicates that `DBGDTRRX_EL0` is empty and the PE is ready to accept a word external write to `DBGDTRRX_EL0`.
- `EDSCR.{TXfull, ITE}` == {1, 1} indicates that `DBGDTRTX_EL0` is full and the PE is ready to accept a word external read from `DBGDTRTX_EL0`.

All other values indicate that the PE is not ready, and result in a DTR overrun or underrun error, an ITR overrun error, or both, as defined in [Overrun and underrun flags](#).

`EDSCR.{ITE, RXfull, TXfull}` shows the status of the ITR and DCC registers. It ignores the question of whether a read or write cannot be accepted because, for example, `EDSCR.ERR` is set or the OPTIONAL Software Lock is locked for memory-mapped accesses (`EDLSR.SLK` == 1).

E4.4.2 Buffering writes to EDITR

The architecture permits a processor to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, provided that:

- Those instructions are discarded if `EDSCR.ERR` is set to 1, either by an underrun or an overrun, or by any other error conditions described in this architecture, such as an instruction generating an abort.
- The PE maintains the simple sequential execution model with the order of instructions determined by the order in which the PE accepts the EDITR writes. In particular, the buffered instructions must be executed in the Execution state consistent with a simple sequential execution of the instructions, even if one of the previous instructions is a state changing operation, such as `DCPS` or `DRPS`.

E4.4.3 Overrun and underrun flags

Each of the ready flags has a corresponding overrun or a corresponding underrun flag. These are sticky status flags that are set if the register is accessed using the external debug interface when the corresponding ready flag is not in the ready state.

If the PE is in Debug state and Memory access mode, the corresponding error flag is also set if the PE is not ready to accept an operation because a previous load or store is still in progress. The sticky status flag remains set until cleared by writing 1 to [EDSCR.CSE](#).

Note

The architecture permits a PE to continue to accept and buffer data to write to memory in Memory access mode.

[Table E4-2](#) shows DCC and ITR ready flags and the overrun and underrun flags associated with them.

Table E4-2 DCC and ITR ready flags and the associated overrun/underrun flags

External debug interface access	Overrun/Underrun condition	EDSCR flag
Write DBGDTRRX_EL0	<code>EDSCR.RXfull == '1' (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')</code>	RXO
Read DBGDTRTX_EL0	<code>EDSCR.TXfull == '0' (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')</code>	TXU
Write EDITR	<code>Halted() && (EDSCR.ITE == '0' EDSCR.MA == '1')</code>	ITO

When an overrun or underrun flag is set to 1, the cumulative error flag, [EDSCR.ERR](#), described in [Cumulative error flag](#), is also set to 1.

In the event of an external write to [DBGDTRRX_EL0](#) or [EDITR](#) generating an overrun, or an external read from [DBGDTRTX_EL0](#) generating an underrun:

- For a write, the written value is ignored.
- For a read, an UNKNOWN value is returned.
- [EDSCR.TXfull](#), [EDSCR.RXfull](#) or [EDSCR.ITE](#), as applicable, are not updated.

There is no overrun or underrun detection on external reads of [DBGDTRRX_EL0](#) or external writes of [DBGDTRTX_EL0](#).

There is no overrun or underrun detection of direct reads and direct writes of the DTR System registers by software:

- If `RXfull == 0`, a direct read of [DBGDTRRXint](#) or [DBGDTR_EL0](#) returns UNKNOWN.
- If `TXfull == 1`, a direct write of:
 - [DBGDTRTXint](#) sets DTRTX to UNKNOWN.
 - [DBGDTR_EL0](#) sets DTRRX and DTRTX to UNKNOWN.

See [DCC accesses in Non-debug state](#) for more information.

E4.4.4 Cumulative error flag

The cumulative error flag, [EDSCR.ERR](#), is set to 1:

- On taking an exception from Debug state.
- On any signaled overrun or underrun in the DCC or ITR.

When [EDSCR.ERR == 1](#):

- External reads of [DBGDTRTX_ELO](#) do not have any side-effects.
- External writes to [DBGDTRRX_ELO](#) are ignored.
- External writes to [EDITR](#) are ignored.
- No further instructions can be issued in Debug state. This includes any instructions previously accepted as external writes to [EDITR](#) that occur in program order after the instruction or access that caused the error.

This allows a debugger to stream data, or, in Debug state, instructions, to the target without having to:

- Check [EDSCR](#).{RXfull, TXfull, ITE} before each access.
- Check [EDSCR](#).{ITO, RXO, TXU} following each access, for overrun or underrun.
- Check PSTATE or other syndrome registers, or both, for an exception following each instruction executed in Debug state that might generate a synchronous exception.

The cumulative error flag remains set until cleared to 0 by writing 1 to [EDRCR.CSE](#). However, the effect of writing 1 to [EDRCR.CSE](#) to clear [EDSCR.ERR](#) is **CONSTRAINED UNPREDICTABLE** when both of the following apply:

- The PE is in Debug state.
- The value of [EDSCR.ITE](#) is 0.

When these conditions apply and a value of 1 is written to [EDRCR.CSE](#), either or both of the following might occur:

- [EDSCR.ERR](#) is not cleared to 0.
- Any instructions in [EDITR](#) that have not been executed might be executed subsequently, rather than being ignored.

Note

This means that a debugger must poll [EDSCR.ITE](#) until it has the value 1, indicating that [EDITR](#) is empty, before writing to [EDRCR.CSE](#) to clear the [EDSCR.ERR](#) flag to 0.

For overruns and underruns, [EDSCR](#).{ITO, RXO, TXU} record the error type.

E4.4.4.1 Pseudocode description of clearing the error flag

The [ClearStickyErrors](#) () pseudocode function is described in [R-profile Architecture Pseudocode](#).

E4.5 Synchronization of DCC and ITR accesses

In addition to the standard synchronization requirements for register accesses, the following subsections describe additional requirements that apply for the DCC and ITR registers:

- [Summary of System register accesses to the DCC.](#)
- [DCC accesses in Non-debug state.](#)
- [Synchronization of DCC interrupt request signals.](#)
- [DCC and ITR access in Debug state.](#)

In these sections, accesses by the external debug interface are referred to as external reads and external writes. Accesses to System registers are referred to as direct reads, direct writes, indirect reads, and indirect writes.

Note

In Synchronization requirements for AArch64 System registers external reads and external writes are described as forms of indirect access. This whole section uses more explicit terminology.

The DTR registers and the DCC flags, TXfull and RXfull, form a communication channel, with one end operating asynchronously to the other. Implementations must respect the ordering of accesses to these registers in order to maintain the correct behavior of the channel.

External reads of, and external writes to [DBGDTRRX_EL0](#) and [DBGDTRTX_EL0](#) are asynchronous to direct reads of, and direct writes to, [DBGDTRRXint](#), [DBGDTRTXint](#), made by software using System register access instructions. The direct reads and direct writes indirectly write to the DCC flags. The external reads and external writes indirectly read the DCC flags to check for underrun and overrun.

Throughout this section:

- | | |
|------------------|--|
| DCC flags | Means any or all of the following: <ul style="list-style-type: none"> • The EDSCR.{RXfull.TXfull} ready flags. • The EDSCR.RXO overrun flag. • The EDSCR.TXU underrun flag. • The EDSCR.ERR cumulative error flag. |
| ITR flags | Means any or all of the following: <ul style="list-style-type: none"> • The EDSCR.ITE ready flag. • The EDSCR.ITO overrun flag. • The EDSCR.ERR cumulative error flag. |

E4.5.1 Summary of System register accesses to the DCC

System register accesses to the DTR registers are direct reads and writes of those registers, as shown in [Table E4-3](#). Several of these instructions access the same registers using different encodings.

[DBGDTRRX_EL0](#) and [DBGDTRTX_EL0](#) are encoded as MRS and MSR accesses respectively to the same System register, even though they access different underlying register values. [DBGDTRRXint](#) and [DBGDTRTXint](#) are similarly encoded as MRC and MCR accesses respectively to the same System register. The encoding means that direct reads and writes using these encodings must be ordered with respect to each other. For more information, see Synchronization requirements for AArch64 System registers and [Synchronization of changes to AArch32 System registers](#).

[Table E4-3](#) shows a summary of System register accesses to the DCC.

Table E4-3 Summary of System register accesses to the DCC

Operation	OS Lock	AArch32 (MRC/MCR)	Description
Read	-	DBGDTRRXint	Direct read of DTRRX. Indirect write to the DCC flags . An STC instruction that reads DBGDTRRXint makes an indirect write to DBGDSCRint.RXfull .
Write	-	DBGDTRTXint	Direct write of DTRTX. Indirect write to the DCC flags . An LDC instruction that writes to DBGDTRTXint using a value read from memory is a direct write to DBGDTRTXint .
Read/write	-	-	Direct read/write of both DTRRX and DTRTX. Indirect write to the DCC flags .
Read	-	DBGDSCRint	Direct read of the DCC flags .
Read/write	-	DBGDTRRXext	Direct read/write of DTRRX.
Read/write	-	DBGDTRTXext	Direct read/write of DTRTX.
Read	Unlocked	DBGDSCRext	Direct read of DCC flags .
Read/write	Locked	DBGDSCRext	Direct read/write of DCC flags .

E4.5.2 DCC accesses in Non-debug state

In Non-debug state DCC accesses are as described in [Normal access mode](#):

- If a direct read of [DBGDSCRint](#) returns $RXfull = 1$, then a following direct read of [DBGDTRRXint](#) returns valid data and indirectly writes 0 to [DBGDSCRint.RXfull](#) as a side-effect.
- If a direct read of [DBGDSCRint](#) returns $TXfull = 0$, then a following direct write to [DBGDTRTXint](#) writes the intended value, and indirectly writes 1 to [DBGDSCRint.TXfull](#) as a side-effect.

No [Context Synchronization event](#) is required between these two instructions. Overrun and underrun detection prevents intervening external reads and external writes affecting the outcome of the second instruction.

The indirect write to the [DCC flags](#) as part of the DTR access instruction is made atomically with the DTR access.

Because a direct read of [DBGDTRRXint](#) is an indirect write to [DBGDSCRint.RXfull](#), it must occur in program order with respect to the direct read of [DBGDSCRint](#), meaning it must not return a speculative value for DTRRX that predates the $RXfull$ flag returned by the read of [DBGDSCRint](#). The direct write to [DBGDTRTXint](#) must not be executed speculatively.

Direct reads of [DBGDTRRXint](#) and [DBGDSCRint](#), must occur in program order with respect to other direct reads of the same register using the same encoding.

The following accesses have an implied order within the atomic access:

- In the simple sequential execution of the program the indirect write of the [DCC flags](#) occurs immediately after the direct DTR access.
- In the simple sequential execution model, for an external read of [DBGDTRRX_EL0](#) or an external write of [DBGDTRTX_EL0](#):
 - The check of the [DCC flags](#) for overrun or underrun occurs immediately before the access.
 - If there is no underrun or overrun, the update of the [DCC flags](#) occurs immediately after the access.
 - If there is underrun or overrun, the update of the DCC underrun or overrun flags occurs immediately after the access.

All observers must observe the same order for accesses.

Note

These requirements do not create order where order does not otherwise exist. It applies only for ordered accesses.

Without explicit synchronization following external writes and external reads:

- The value written by the external write to [DBGDTRRX_EL0](#) that does not overrun, must be observable to direct reads of [DBGDTRRXint](#) in finite time.
- The [DCC flags](#) that are updated as a side-effect of the external write or external read must be observable:
 - To subsequent external reads of [EDSCR](#).
 - To subsequent external reads of [DBGDTRRX_EL0](#) when checking for underrun.
 - To subsequent external writes to [DBGDTRTX_EL0](#) when checking for overrun.
 - To direct reads of [DBGDSCRint](#) in finite time.

However, explicit synchronization is required to guarantee that a direct read of [DBGDSCRint](#) returns up-to-date [DCC flags](#). This means that if a signal is received from another agent that indicates that [DBGDSCRint](#) must be read, an [ISB](#) is required to ensure that the direct read of [DBGDSCRint](#) occurs after the signal has been received. This also synchronizes the value in [DBGDTRRXint](#), if applicable. However, if that signal is an interrupt exception triggered by [COMMIQ](#), [COMMTX](#), or [COMMRX](#), the exception entry is sufficient synchronization. See [Synchronization of DCC interrupt request signals](#).

Explicit synchronization is required following a direct read or direct write:

- To ensure that a value directly written to [DBGDTRTXint](#) is observable to external reads of [DBGDTRTX_EL0](#).
- To ensure that a value directly written to [DBGDTR_EL0](#) is observable to external reads of [DBGDTRTX_EL0](#) and [DBGDTRRX_EL0](#).
- To guarantee that the indirect writes to the [DCC flags](#) that were a side-effect of the direct read or direct write have occurred, and therefore that the updated values are:
 - Observable to external reads of [EDSCR](#).
 - Observable to external reads of [DBGDTRRX_EL0](#) when checking for underrun.
 - Observable to external writes of [DBGDTRTX_EL0](#) when checking for overrun.
 - Returned by a following direct read of [DBGDSCRint](#).

See also [Memory-mapped accesses to the DCC and ITR](#) and [Synchronization of changes to the external debug registers](#).

Note

These ordering rules mean that software:

- Must not read [DBGDTRRXint](#) or [DBGDTR_EL0](#) without first checking [DBGDSCRint.RXfull](#) or if the previously-read value of [DBGDSCRint.RXfull](#) is 0.
It is not sufficient to read both registers and then later decide whether to discard the read value, as there might be an intervening write from the external debug interface.
 - Must not write [DBGDTRTXint](#) or [DBGDTR_EL0](#) without first checking [DBGDSCRint.TXfull](#) or if the previously-read value of [DBGDSCRint.TXfull](#) is 1.
The write to [DBGDTRTXint](#) overwrites the value in [DTRTX](#), and the external debugger might or might not have read this value.
 - Must ensure there is an explicit [Context Synchronization event](#) following a DTR access, even if not immediately returning to read [DBGDSCRint](#) again. This synchronization operation might be an exception return.
-

E4.5.2.1 Derived requirements

The rules for DCC accesses in Non-debug state are as follows:

- Following a direct read of [DBGDTRRXint](#) when [RXfull](#) is 1:
 - If an external write to [DBGDTRRXint](#) checks the [RXfull](#) flag for overrun and observes that the value of [RXfull](#) is 0, the value returned by the previous direct read must not be affected by the external write.

- If an external read of [EDSCR](#) returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by a following external write to [DBGDTRRXint](#), and the following external write does not overrun.
- Following a direct write to [DBGDTRTXint](#), when TXfull is 0:
 - If an external read of [DBGDTRTXint](#) checks the TXfull flag for underrun and observes that the value of TXfull is 1, the value returned by the external read must be the value written by the previous direct write.
 - If an external read of [EDSCR](#) returns a TXfull value of 1, then the value returned by a following external read of [DBGDTRRXint](#) must be the value written by the previous direct read, and the subsequent external read will not underrun.
- Following an external read of [DBGDTRTXint](#) that does not underrun, if a direct read of [DBGDSCRint](#) returns a TXfull value of 0, then the value returned by the external read must not be affected by a following direct write to [DBGDTRTXint](#).
- Following a first external read [DBGDTRRXint](#) and a following second external read of [DBGDTRTXint](#) that does not underrun, if a direct read of [DBGDSCRint](#) returns a TXfull value of 0, then the values returned by the external reads must not be affected by a following direct write to [DBGDTR_EL0](#).
- Following an external write to [DBGDTRRXint](#) that does not overrun, if a direct read of [DBGDSCRint](#) returns an RXfull value of 1, then the value returned by a following direct read of [DBGDTRRXint](#) or [DBGDTR_EL0](#) must be the value written by the previous external write.
- Following a first external write to [DBGDTRTXint](#) and a following second external write to [DBGDTRRXint](#) that does not overrun, if a direct read of [DBGDSCRint](#) returns an RXfull value of 1, then the value returned by a subsequent direct read of [DBGDTR_EL0](#) must return the values written by the previous external writes.

E4.5.3 Synchronization of DCC interrupt request signals

Following an external read or external write access to the DTR registers, the interrupt request signals, **COMMIRQ**, **COMMTX**, and **COMMRX**, must be updated in finite time without explicit synchronization.

The updated values must be observable to a direct read of [DBGDSCRint](#) or [DBGDTRRXint](#), or a direct write of [DBGDTRTXint](#) executed after taking an interrupt exception generated by the interrupt request. The updated values must also be observable to a direct write of [DBGDTRTXint](#) executed after taking an interrupt exception generated by the interrupt request.

Note

The requirement that indirect writes to registers are observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time. For more information, see [Synchronization of changes to AArch32 System registers](#).

Following a direct read of [DBGDTRRXint](#) or a direct write to [DBGDTRRXint](#), software must execute a [Context Synchronization event](#) to guarantee the interrupt request signals have been updated in finite time. This synchronization operation can be an exception return.

E4.5.4 DCC and ITR access in Debug state

In Debug state, stricter observability rules apply for instructions issued through the ITR, to maintain communication between a debugger and the PE, without requiring excessive explicit synchronization.

In Normal access mode, without explicit synchronization:

- A direct read or direct write of the DTR registers by an instruction written to [EDITR](#) must be observable to an external write or an external read in finite time:
 - A direct read of [DBGDTRRXint](#) must be observable to an external write of [DBGDTRRX_EL0](#).
 - A direct read of [DBGDTR_EL0](#) must be observable to an external write of [DBGDTRRX_EL0](#) and [DBGDTRTX_EL0](#).
 - A direct write of [DBGDTRTXint](#) must be observable to an external read of [DBGDTRTX_EL0](#).
 - A direct write of [DBGDTR_EL0](#) must be observable to an external read of [DBGDTRRX_EL0](#) and [DBGDTRTX_EL0](#).

This includes the indirect write to the [DCC flags](#) that occurs atomically with the access as described in [DCC accesses in Non-debug state](#).

The subsequent external write or external read must observe either the old or the new values of both the DTR contents and [DCC flags](#). If the old values are observed, this typically results in overrun or underrun, assuming the old values of the [DCC flags](#) indicate an overrun or underrun condition, as would normally be the case.

This means the debugger can observe the direct read or direct write without explicit synchronization and without explicitly testing the [DCC flags](#) in [EDSCR](#), because it can rely on overrun and underrun tests.

- External reads of [DBGDTRTX_EL0](#) that do not underrun and external writes to [DBGDTRRX_EL0](#) that do not overrun must be observable to an instruction subsequently written to [EDITR](#) on completion of the first external access. This includes the indirect write to the [DCC flags](#).

This means that without explicit synchronization and without the need to first check the [DCC flags](#) in [DBGDSCRint](#):

- If the instruction is a direct read of [DBGDTRRXint](#), it observes the external write.
- If the instruction is a direct write of [DBGDTRTXint](#), it observes the external read.
- Writes to [EDITR](#) that do not overrun commit an instruction for execution immediately. The instruction must complete execution in finite time without requiring any further operation by the debugger.
- After an external write to the [EDITR](#), the [ITR flags](#) that are updated as a side effect of that write must be observable by:
 - An external read of the [EDSCR](#) that follows the external write to the [EDITR](#).
 - When checking for overrun, another external write to the [EDITR](#) that follows the original external write to the [EDITR](#).

In Memory access mode, these requirements shift to the instructions implicitly executed by external reads and external writes of the DTR registers, as described in [Memory access mode](#).

E4.6 Interrupt-driven use of the DCC

Arm recommends implementations provide a level-sensitive DCC interrupt request through the IMPLEMENTATION DEFINED interrupt controller as a Private Peripheral Interrupt for the originating PE.

Note

- In addition to connection to the interrupt controller Arm also recommends **COMMIRQ**, **COMMTX**, and **COMMRX** signals that might be implemented for use by any legacy system peripherals.
 - GICv3 reserves a Private Peripheral Interrupt number for the **COMMIRQ** interrupt.
-

The [DBGDCCINT](#) register provides a first level of interrupt masking within the PE, meaning only a single interrupt source, **COMMIRQ**, is needed at the interrupt controller.

See also [Synchronization of DCC interrupt request signals](#).

E4.7 Pseudocode description of the operation of the DCC and ITR registers

The basic operation of the DCC and ITR registers is shown by the following pseudocode functions. These functions do not cover the behavior when `DBGOSLSR.OSLK == 1`, meaning that the OS Lock is locked:

- `Write_DBGDTR_EL0()`.
- `Write_DBGDTRRX_EL0()`.
- `Write_DBGDTRTX_EL0()`.
- `Write_EDITR()`.
- `CheckForDCCInterrupts()`.

For the definition of the DTR Registers, see [shared/debug/dccanditr/DTR](#).

Chapter E5

The Embedded Cross-Trigger Interface

This chapter describes the Embedded Cross-Trigger interface. It contains the following sections:

- [About the Embedded Cross-Trigger](#)
- [Basic operation on the ECT](#)
- [Cross-triggers on a PE in an Arm R-profile implementation](#)
- [Description and allocation of CTI triggers](#)
- [CTI registers programmers' model](#)
- [Examples](#)

E5.1 About the Embedded Cross-Trigger

The *Embedded Cross-Trigger* (ECT) allows a debugger to:

- Send trigger events to a PE. For example, this might be done to halt the PE.
- Send a trigger event to one or more PEs, or other system components, when a trigger event occurs on another PE or system component. For example, this might be done to halt all PEs when one individual PE halts.

Figure E5-1 shows the logical structure of an ECT.

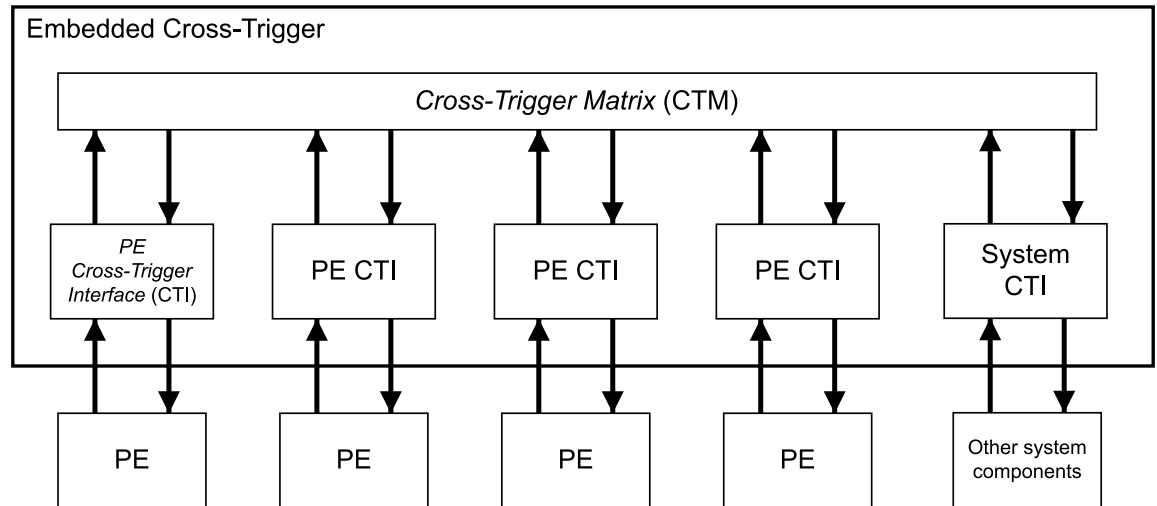


Figure E5-1 Structure of an Embedded Cross-Trigger

The ECT can deliver many types of trigger events, which are described in the following sections:

- [Debug request trigger event.](#)
- [Restart request trigger event.](#)
- [Cross-halt trigger event.](#)
- [Performance Monitors overflow trigger event.](#)
- [Generic trace external input trigger events.](#)
- [Generic trace external output trigger events.](#)
- Generic CTI interrupt trigger event.

From the introduction of Armv8, an implementation must:

- Include a Cross-Trigger interface, CTI.
- Implement at least the input and output triggers defined in this architecture.

In addition, see .

Arm recommends that this Cross-Trigger interface includes:

- The ability to route trigger events between Trace Units, which typically have advanced event triggering logic.
- An output trigger to the interrupt controller.

Also, Arm recommends that the Embedded Cross-Trigger includes the capability to send and receive IMPLEMENTATION DEFINED system trigger events to and from other system components, including a system counter, using a system CTI. See [Halt-on-debug](#).

Note

The ECT and CTI must only signal trigger events for external debugging. They must not route software events, such as interrupts. For example, the Performance Monitors overflow input trigger is provided to allow entry to Debug state on a counter overflow, and the output trigger to the interrupt controller is provided to generally allow events from the external debug subsystem to be routed to a software agent. However, the combination of the two must not be used as a mechanism to route Performance Monitors overflows to an interrupt controller.

Note

CTI version 1 (CTIv1) is defined by the *CoreSight™ SoC Technical Reference Manual*. CTIv2 extends CTIv1 with the addition of the input channel gate. See Implementation with CTIv2.

E5.1.1 Implementation with a CoreSight CTI

For details of the recommended connections for an A-profile implementation, see Recommended External Debug Interface. See also *CoreSight™ SoC Technical Reference Manual*.

E5.2 Basic operation on the ECT

The ECT comprises a Cross-Trigger Matrix, CTM, and one Cross-Trigger Interface, CTI, for each PE. The ECT might also include other CTIs for other system components. The CTM passes events between the CTI blocks over channels. The CTM can have a maximum of 32 channels.

The main interfaces of the Cross-Trigger interface, CTI, are:

- The input triggers:
 - These are trigger event inputs from the PE to the CTI.
- The output triggers:
 - These are trigger event outputs from the CTI to the PE.
- The input channels:
 - These are channel event inputs from the Cross-Trigger matrix, CTM, to the CTI.
- The output channels:
 - These are channel event outputs from the CTI to the CTM.

Each CTI block has:

- Up to 32 input triggers that come from the PE:
 - The input triggers are numbered 0-31.
- Up to 32 output triggers that go to the PE:
 - The output triggers are numbered 0-31.

If the CTI is not powered up when the Core power domain is powered up, the CTI ignores all input triggers and input channel events, and does not generate any output triggers or output channel events.

[Figure E5-2](#) shows the logical internal structure of a CTI.

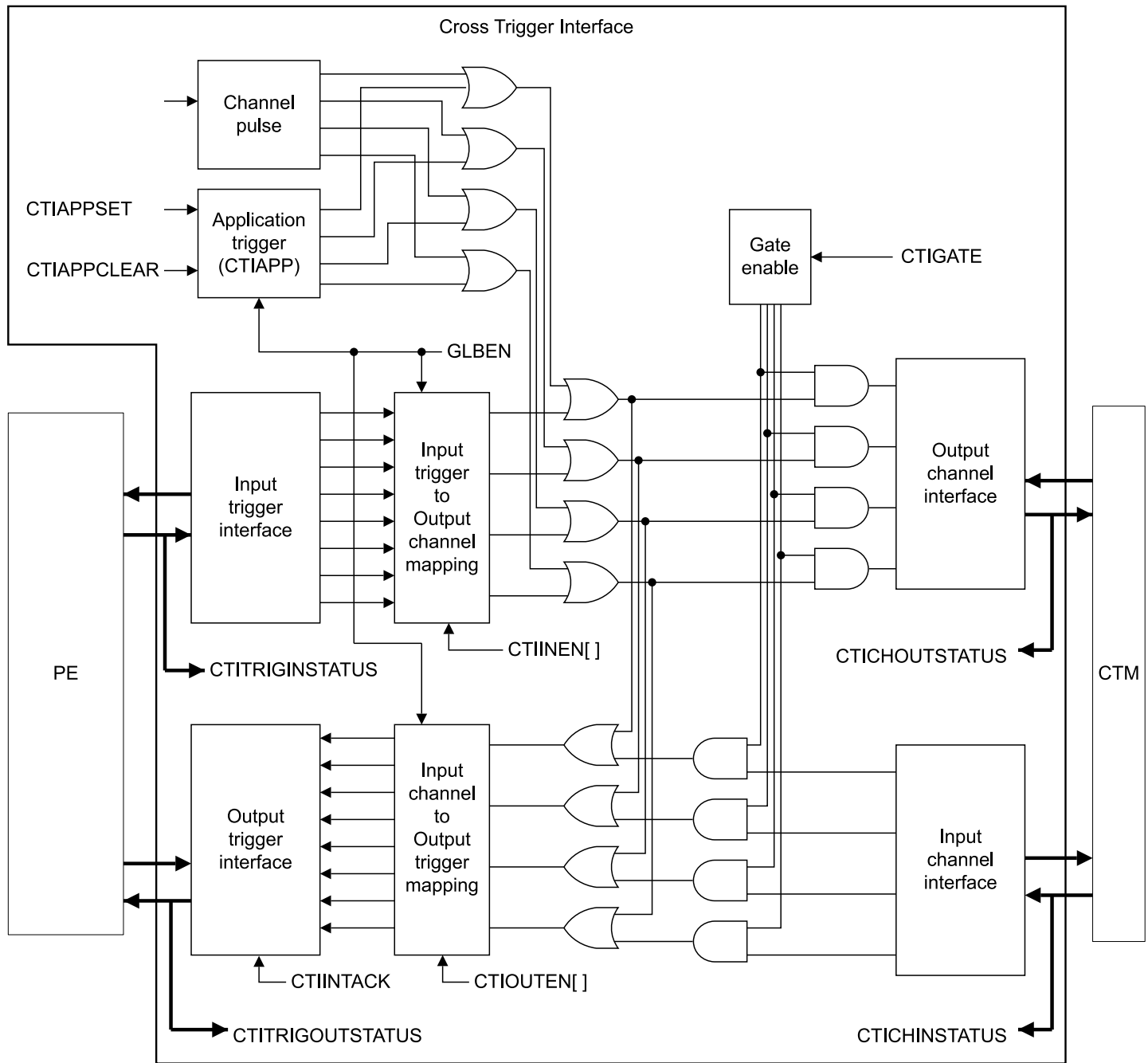


Figure E5-2 Structure of a Cross-Trigger interface

Note

- The number of triggers is IMPLEMENTATION DEFINED. Figure E5-2 shows eight input and eight output triggers.
- The number of channels is IMPLEMENTATION DEFINED. Figure E5-2 shows four channels.
- In Figure E5-2 the input channel gate function is a CTIv2 feature.

When the CTI receives an input trigger event, this generates channel events on one or more internal channels, according to the *mapping function* defined by the *Input trigger→output channel mapping registers*, *CTIINEN<n>*.

The CTI also contains an *application trigger* and *channel pulse* to allow a debugger to create channel events directly on internal channels by writing to the CTI control registers.

Channel events on each internal channel are passed to a corresponding output channel that is controlled by a *channel gate*. The channel gate can block propagation of channel events from an internal channel to an output channel.

Note

If the CTM is implemented:

- The gate function must be implemented.
 - If the CTI is CTIv1, the gate function applies to output triggers only.
-

The output channels from a CTI are combined, using a logical OR function, with the output channels from all other CTIs to form the input channels on other CTIs. The input channels of this CTI are the logical OR of the output channels on all other CTIs. This is the *Cross-Trigger Matrix*, CTM. Therefore, the number of input channels must equal the number of output channels.

Note

The number of input triggers and output triggers is not required to be the same.

The internal channels form an internal Cross-Trigger matrix within the CTI. This delivers events directly from the input triggers to the output triggers. Therefore the number of internal channels is the same as the number of input and output channels on the external CTM, and there is a direct mapping between the two.

Channel events received on each input channel are passed to the corresponding internal channel. It is IMPLEMENTATION DEFINED whether the Cross-Trigger gate also blocks propagation of channel events from input channels to internal channels.

Note

If CTIv2 is implemented, the Cross-Trigger gate also blocks propagation of channel events from input channels to internal channels.

When the CTI receives a channel event on an internal channel this generates trigger events on one or more output triggers, according to the mapping function defined by the *Input channel* → *output trigger mapping registers*, [CTIOUTEN<n>](#).

The CTI contains the input and output trigger interfaces to the PE and the interface of the Cross-Trigger matrix. The architecture does not define the signal protocol used on the trigger interfaces, and:

- It is IMPLEMENTATION DEFINED whether the CTI supports multicycle input trigger events.
- It is IMPLEMENTATION DEFINED whether the CTM supports multicycle channel events.

See [Multicycle events](#).

However, an output trigger is asserted until acknowledged. The output trigger can be:

- Self-acknowledging. This means that no further action is required from the debugger.
- Acknowledged by the debugger writing 1 to the corresponding bit of [CTIINTACK](#).

The time taken to propagate a trigger event from the first PE, through its CTI, across the CTM to another CTI, and thereby to a second PE is IMPLEMENTATION DEFINED.

Note

Arm recommends that this path is not longer than the shortest software communication path between those PEs. This is because if the first PE halts, the Cross-halt trigger event can propagate through the ECT and halt the second PE without causing software on the second PE to malfunction because the first PE is in Debug state and is not responding.

E5.2.1 Multicycle events

A multicycle event is one with a continuous state that might persist over many cycles, as opposed to a discrete event. A typical implementation of a multicycle event is a level-based signal interface, whereas a discrete event might be implemented as a pulse signal or message.

CTI support for multicycle trigger events is IMPLEMENTATION DEFINED. Use of multicycle trigger events is deprecated. Of the architecturally defined input trigger events, the Performance Monitors overflow trigger event and Generic trace external output trigger events can be multicycle input triggers.

CTM support for multicycle channel events is IMPLEMENTATION DEFINED. A CTM that does not support multicycle channel events cannot propagate a multicycle trigger event between CTIs.

Note

A full ECT might comprise a mix of CTIs, some of which can support multicycle trigger events. In bridging these components, multicycle channel events become single channel events at the boundary between the CTIs.

E5.2.1.1 An ECT that supports multicycle trigger events

When an ECT supports multicycle trigger events, an input trigger event to the CTI continuously asserts channel events on all output channels mapped to it until either:

- The input trigger event is removed.
- The channel mapping function is disabled.

This means that an input trigger that is asserted for multiple cycles causes any channels that are mapped to it to become active for multiple cycles. Consequently, any output triggers mapped from that channel are asserted for multiple cycles.

Note

The output trigger remains asserted for at least as long as the channel remains active. This means that even if the output trigger is acknowledged, it remains asserted until the channel deactivates.

The CTI does not guarantee that these events have precisely the same duration, as the triggers and channels can cross between clock domains.

[CTIAPPSET](#) and [CTIAPPCLEAR](#) can set a channel active for multiple cycles. [CTIAPPULSE](#) generates a single channel event. [CTICHINSTATUS](#) and [CTICHOUTSTATUS](#) can report whether a channel is active.

E5.2.1.2 An ECT that does not support multicycle trigger events

When an ECT does not support multicycle trigger events, an input trigger event to the CTI generates a single channel event on all output channels mapped to it, regardless of how long the input trigger event is asserted.

This means that an input trigger event that is asserted for multiple cycles generates a single channel event on any channels mapped to it. Consequently any self-acknowledging output triggers mapped from those channels are single trigger events.

Note

A single event is typically a single cycle, but there is no guarantee that this is always the case.

[CTIAPPSET](#) and [CTIAPPCLEAR](#) can only generate a single channel event. [CTIAPPULSE](#) generates a single channel event. If the ECT does not support multicycle channel events, use of [CTIAPPSET](#) and [CTIAPPCLEAR](#) is deprecated, and the debugger must only use [CTIAPPULSE](#). [CTICHINSTATUS](#) and [CTICHOUTSTATUS](#) must be treated as UNKNOWN.

E5.3 Cross-triggers on a PE in an Arm R-profile implementation

An R-profile PE must include a Cross-Trigger interface, and the implementation must include at least the input and output triggers defined in this architecture. The number of channels in the Cross-Trigger matrix is IMPLEMENTATION DEFINED, but there must be a minimum of three. Software can read `CTIDEVID.NUMCHAN` to discover the number of implemented channels.

The CTM must connect to all PEs in the same Inner Shareability domain as the PE that includes the Cross-Trigger interface, but can also connect to additional PEs. Arm strongly recommends that the CTM connects all PEs implementing a CTI in the system. This includes other PEs that can be connected using a CoreSight CTI module.

Note

In a uniprocessor system the CTM is OPTIONAL. In a multiprocessor system the CTM is required. The CTM might be connected other CTI modules for non-PEs, such as triggers for system visibility components. Arm recommends that the CTM is implemented.

Any CTI connected to a PE that is not an Arm PE must implement at least:

- The Debug request trigger event.
- The Restart trigger event.
- The Cross-halt trigger event.

For more information about the CTI, see the *CoreSight™ SoC Technical Reference Manual*. Arm architecture refines the generic CTI by defining roles for each of the implemented input and output triggers.

E5.4 Description and allocation of CTI triggers

Table E5-1 shows the output trigger events defined by the architecture and the related trigger numbers.

Table E5-1 Allocation of CTI output trigger events

Number	Source	Destination	Event description
0	CTI	PE	Debug request trigger event
1	CTI	PE	Restart request trigger event
2	CTI	IRQ controller	Generic CTI interrupt trigger event
3	-	-	Reserved
4 - 7	CTI	Trace unit	OPTIONAL Generic trace external input trigger events
8 - 15	-	-	Reserved

Note
Output triggers from the CTI are inputs to other blocks.

Table E5-2 shows the input trigger events defined by the architecture and the related trigger numbers.

Table E5-2 Allocation of CTI input trigger events

Number	Source	Destination	Event description
0	PE	CTI	Cross-halt trigger event
1	PE	CTI	Performance Monitors overflow trigger event
2	-	-	Reserved
3	-	-	Reserved
4 - 7	Trace unit	CTI	OPTIONAL Generic trace external output trigger events
8 - 10	-	-	Reserved
11 - 15	-	-	IMPLEMENTATION DEFINED.

Note
Input triggers to the CTI are outputs from other blocks.

Table E5-1 and Table E5-2 show the minimum set of trigger events defined by the architecture. However:

- The Generic trace external input and output trigger events are required only if the OPTIONAL trace unit is implemented. If the OPTIONAL trace unit is not implemented, these trigger events are reserved.
- Support for the generic CTI interrupt trigger event is IMPLEMENTATION DEFINED because details of interrupt handling in the system, including any interrupt controllers, are IMPLEMENTATION DEFINED. Details regarding how the CTI interrupt is connected to an interrupt controller and its allocated interrupt number lie outside the scope of the architecture. Arm strongly recommends that implementations provide a means to generate interrupts based on external debug events.
- The other trigger events are required by the architecture.

E5.4.1 Debug request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to force the PE into Debug state. The trigger event is asserted until acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to [CTIINTACK](#)[0].

Note

A debugger must poll [CTITRIGOUTSTATUS](#)[0] until it reads as 0, to confirm that the output trigger has been deasserted before generating any event that must be ordered after the write to [CTIINTACK](#), such as a write to [CTIAPPPULSE](#) to activate another trigger.

If the PE is already in Debug state, the PE ignores the trigger event, but the CTI continues to assert it until it is removed by the debugger. See also [External Debug Request debug event](#).

E5.4.2 Restart request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to request the PE to exit Debug state. If the PE is in Non-debug state, the request is ignored by the PE.

If a Restart request trigger event is received at or about the same time as the PE enters Debug state, it is CONstrained UNPREDICTABLE whether:

- The request is ignored by the PE. In this case the PE enters Debug state and remains in Debug state.
- The PE enters Debug state and then immediately restarts.

Debuggers must program the CTI to send Restart request trigger events only to PEs that are halted. To enable the PE to disambiguate discrete Restart request trigger events, after sending a Restart request trigger event, the debugger must confirm that the PE has restarted and halted before sending another Restart request trigger event. Debuggers can use [EDPRSR](#).{SDR, HALTED} to determine the Execution state of the PE.

Note

Before generating a Restart request trigger event for a PE, a debugger must ensure any Debug request trigger event targeting that PE is cleared. [Debug request trigger event](#) describes how to do this.

The trigger event is self-acknowledging, meaning that the debugger requires no further action to remove the trigger event. The trigger event is acknowledged even if the request is ignored by the PE. See also [Exiting Debug state](#).

E5.4.3 Cross-halt trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted by a PE when it is entering Debug state.

Note

To reduce the latency of halting, Arm recommends that an implementation issues the Cross-halt trigger event early in the committed process of entering Debug state. This means that there is no requirement to wait until all aspects of entry to Debug state have completed before issuing the trigger event. Speculative emission of Cross-halt trigger events is not allowed. The Cross-halt trigger event must not be issued early enough for a subsequent Debug request trigger event, which might be derived from the Cross-halt trigger event, to be recorded in the [EDSCR](#).STATUS field. This applies to Debug request trigger events that are acting as inputs to the PE.

E5.4.4 Performance Monitors overflow trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted each time the PE records a Performance Monitors counter overflow. See [The Performance Monitors Extension](#).

If the CTI supports multicycle trigger events, then the trigger event remains asserted while any of the following are true, and deasserted otherwise:

- An event counter <n> is implemented, the global enable control for the event counter is 1, `PMINTENSET.P[n]` is 1, and `PMOVSSSET.P[n]` is 1. The global enable control is defined in [Enabling PMU counters](#).
- `PMCR.E` is 1, `PMINTENSET.C` is 1, and `PMOVSSSET.C` is 1.

Note

- This does not replace the recommended connection of Performance Monitors overflow trigger event to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI.
 - Events can be counted when `ExternalNoninvasiveDebugEnabled()` == FALSE, and, in Secure state, when `ExternalSecureNoninvasiveDebugEnabled()` == FALSE. Secure software must be aware that overflow trigger events are nevertheless visible to the CTI.
-

If the CTI does not support multicycle trigger events, then the trigger event is asserted whenever a counter overflows, causing the PE to set `PMOVSSSET.P[n]` to 1, or a direct write to a System register means the above conditions become true.

For more information, see [An ECT that supports multicycle trigger events](#).

E5.4.5 Statistical Profiling Extension sample trigger event

If the Statistical Profiling Extension is implemented, and a sample record is written to memory, CTI input trigger 2 is asserted. This trigger might also be directly connected to other IMPLEMENTATION DEFINED debug features.

For more information, see The Statistical Profiling Extension.

E5.4.6 Generic trace external input trigger events

These are output trigger events from the CTI, and input trigger events to the OPTIONAL trace unit, that are used in conjunction with the Generic trace external output trigger events to pass trigger events between:

- The PE and the OPTIONAL trace unit.
- The OPTIONAL trace unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external input trigger events.

The trigger events are self-acknowledging. This means that the debugger does not have to take any further action to remove the events.

E5.4.7 Generic trace external output trigger events

These are input trigger events to the CTI, and output trigger events from the OPTIONAL trace unit, used in conjunction with the Generic trace external input trigger events to pass trigger events between:

- The PE and the OPTIONAL trace unit.
- The OPTIONAL trace unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external output trigger events.

E5.5 CTI registers programmers' model

The CTI registers programmers' model is described in [About the External Debug Registers](#). The following sections contain information specific to the CTI:

- [External debug register resets](#).
- [External debug interface register access permissions](#).
- [Cross-trigger interface registers](#).
- The individual register descriptions in [Cross-Trigger Interface registers](#).

See also [Memory-mapped accesses to the external debug interface](#).

E5.5.1 CTI reset

An External Debug reset resets the CTI. See [External debug register resets](#) for details of CTI register resets. All CTI output triggers and output channels are deasserted on an External Debug reset.

Note

An indirect read of an output trigger might not observe the deasserted state until the processor is Cold reset. For more information, see [Synchronization of changes to the external debug registers](#).

E5.5.2 CTI authentication

The CTI ignores the state of the IMPLEMENTATION DEFINED authentication interface. This means that:

- [CTITRIGINSTATUS](#) shows the status of the input triggers and [CTICHINSTATUS](#) shows the status of the input channels, regardless of the value of [ExternalNoninvasiveDebugEnabled\(\)](#).

Note

The PE does not generate the Cross-halt trigger event and the trace unit does not generate Generic trace external output trigger events when [ExternalNoninvasiveDebugEnabled\(\)](#) == FALSE. However, the PE can generate Performance Monitors overflow trigger events.

- The CTI can generate external triggers regardless of the value of [ExternalInvasiveDebugEnabled\(\)](#) and [ExternalNoninvasiveDebugEnabled\(\)](#).

Note

The destination of the trigger is responsible for authenticating the trigger request:

- The PE ignores Debug request events when [ExternalInvasiveDebugEnabled\(\)](#) is FALSE.
 - It is IMPLEMENTATION DEFINED whether the PE ignores Restart request trigger events when [ExternalInvasiveDebugEnabled\(\)](#) is FALSE.
 - The trace unit ignores Generic trace external input trigger events when [ExternalNoninvasiveDebugEnabled\(\)](#) is FALSE.
 - The behavior of Generic CTI interrupt requests is part of the IMPLEMENTATION DEFINED handling of these interrupts, but it is permissible for an interrupt controller to receive these requests even when [ExternalInvasiveDebugEnabled\(\)](#) is FALSE, meaning software might need to configure the interrupt controller to ignore these requests.
-

E5.6 Examples

The CTI is fully programmable and allows for flexible cross-triggering of events within a PE and between PEs in a multiprocessor system. For example:

- The Cross-halt trigger event and the Debug request trigger event can be used for cross-triggering in a multiprocessor system.
- The Cross-halt trigger event and the Generic interrupt trigger event can be used for event-driven debugging in a multiprocessor system.
- The Performance Monitors overflow trigger event and the Debug request trigger event can force entry to Debug state on overflow of a Performance Monitors event counter, for event-driven profiling.

Note

This does not replace the recommended connection of Performance Monitors overflow trigger events to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI. Arm recommends that the Performance Monitors overflow signal is directly available as a local interrupt source.

- The Generic trace external input and Generic trace external output trigger events can pass trace events into and out of the event logic of the trace unit. They can do this:
 - To pass trace events between Trace Units.
 - In conjunction with the Performance Monitors overflow trigger event, to couple the Performance Monitors to the PE trace unit.
 - In conjunction with the Debug request trigger event, to trigger entry to Debug state on a trace event.
 - In conjunction with other CTIs, to signal a trace trigger event onto a CoreSight trace interconnect.

The following sections describe some examples in more detail:

- [Example E5-1.](#)
- [Example E5-2.](#)
- [Example E5-3.](#)
- [Example E5-4.](#)

Example E5-1 Halting a single PE

To halt a single PE, set:

1. [CTIGATE](#)[0] to 0, so that the CTI does not pass channel events on internal channel 0 to the CTM.
2. [CTIOUTEN0](#)[0] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 0.

Note

The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instance of [CTIOUTEN<n>](#) for which <n> is 0.

3. [CTIAPPULSE](#)[0] to 1, to generate a channel event on channel 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to [CTIINTACK](#)[0], before restarting the PE.

Example E5-2 Halting all PEs in a group when any one PE halts

To program a group of PEs so that when one PE in the group halts, all of the PEs in that group halt, set the following registers for each PE in the group:

1. **CTIGATE**[2] to 1, so that each CTI passes channel events on internal channel 2 to the CTM.
2. **CTIINEN0**[2] to 1, so that each CTI generates a channel event on channel 2 in response to a Cross-halt trigger event.
3. **CTIOUTEN0**[2] to 1, so that each CTI generates a Debug request trigger event in response to a channel event on channel 2.

Note

The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instances of **CTIINEN**<n> and **CTIOUTEN**<n> for which <n> is 0.

When a PE has halted, clear the Debug request trigger event by writing a value of 1 to **CTIINTACK**[0], before restarting the PE.

Example E5-3 Synchronously restarting a group of PEs

To restart a group of PEs, for each PE in the group:

1. If the PE was halted because of a Debug request trigger event, the debugger must ensure the trigger event is deasserted. It can do this by:
 - a. Writing 1 to **CTIINTACK**[0] to clear the Debug request trigger event.
 - b. Polling **CTITRIGOUTSTATUS**[0], until it reads as 0, to confirm that the trigger event has been deasserted.
2. Set **CTIGATE**[1] to 1, so that each CTI passes channel events on internal channel 1 to the CTM.
3. Set **CTIOUTEN1**[1] to 1, so that each CTI generates a Restart request trigger event in response to a channel event on channel 1.

Note

This example must use the instance of **CTIOUTEN**<n> for which <n> is 1.

4. Set **CTIAPPPULSE**[1] to 1 on any one PE in the group, to generate a channel event on channel 1.

Example E5-4 Halting a single PE on Performance Monitors overflow

To halt a single PE on a Performance Monitors overflow set:

1. `CTIGATE[3]` to 0, so that the CTI does not pass channel events on internal channel 3 to the CTM.
2. `CTIINEN1[3]` to 1, so that the CTI generates a channel event on channel 3 in response to a Performance Monitors overflow trigger event.

Note

This step of this example must use the instance of `CTIINEN<n>` for which `<n>` is 1.

3. `CTIOUTEN0[3]` to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 3.

Note

This step of this example must use the instance of `CTIOUTEN<n>` for which `<n>` is 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to `CTIINTACK[0]`, before restarting the PE. Clear the overflow status by writing to `PMOVSCLR_EL0`.

Chapter E6

Debug Reset and Powerdown Support

This chapter describes the reset and powerdown support in the Debug architecture. It contains the following sections:

- [About Debug over powerdown](#)
- [Power domains and debug](#)
- [Core power domain power states](#)
- [Powerup request mechanism](#)
- [Emulating low-power states](#)
- [Debug OS Save and Restore sequences](#)
- [Reset and debug](#)

E6.1 About Debug over powerdown

Arm A-profile external debug defines a logical model for the hardware on which a PE executes. This hardware is logically split into the *Core power domain* and the *Debug power domain*, and the model contains descriptions of the states of those domains. See:

- [Power domains and debug.](#)
- [Core power domain power states.](#)

An implementation may allow power domains to be powered up and down independently. Debug over powerdown provides:

- A facility for software executing on the PE to save and restore the PE state on behalf of a self-hosted or external debugger or both. See [Debug OS Save and Restore sequences.](#)
- A facility for an external debugger to request power up of the Core power domain. See [Powerup request mechanism.](#)
- A facility for an external debugger, or software executing on the PE, to request emulation of powerdown of the Core power domain. See [Emulating low-power states.](#)

E6.2 Power domains and debug

Arm A-profile external debug has two logical power domains, each with its own reset:

- The Debug power domain contains the interface between the PE and the external debugger, and is powered up whenever an external debugger is connected to the SoC. It remains powered up while the external debugger is connected. When the Core power domain is completely off or in a low-power state, a debugger is permitted to access a register that is implemented in the Debug power domain. Registers in this domain are reset by an External Debug reset.
- The Core power domain contains the rest of the PE, and might be allowed to power up and power down independently of the Debug power domain.

Note

- The model of two logical power domains has an impact on the reset and access permission requirements of the debug programmers' model.
 - The power domains are described as logical because the architecture defines the requirements but does not require two physical power domains. Any power domain split that meets the requirements of the programmers' model is a valid implementation.
-

The Core power domain contains several types of registers:

- Non-debug logic refers to all registers and logic that are not associated with debug.
- Self-hosted debug logic refers to registers and logic associated solely with the self-hosted debug aspects of the architecture.
- Shared debug logic refers to registers and logic associated with both the self-hosted and external debug aspects of the architecture.
- External debug logic refers to registers and logic associated solely with the external debug aspects of the architecture.

For information about which groups of registers and components are in each power domain, and which registers change power domain if FEAT_DoPD is implemented, see:

- [Access permissions for the External debug interface registers.](#)
- [Cross-trigger interface registers.](#)
- Management register access permissions.
- [Access permissions for external views of the Performance Monitors.](#)

E6.3 Core power domain power states

The Arm architecture does not define the power states of the PE as these are not normally visible to software. However, they are visible to the external debugger. Arm A-profile external debug uses a four logical power states model for the Core power domain. Mechanisms for entering a low-power state describes the architectural mechanisms for entering low-power states.

When the PE enters a low-power state other than Powerdown by executing an instruction, it remains in that low-power state until it receives a wakeup event. See Mechanisms for entering a low-power state for the definition of wakeup events. When halting is allowed, an External Debug Request is a wakeup event. In addition, if the PE enters the Standby state for any reason, it will leave that state to service an External Debug Request debug event.

The four logical power states are as follows:

Normal	The Core power domain is fully powered up and the debug registers are accessible.
Standby	The Core power domain is on, but there are measures to reduce energy consumption. There can be other IMPLEMENTATION DEFINED measures the OS can take to enter standby. The PE preserves the PE state, including the debug logic state. Changing from standby to normal operation does not involve a reset of the PE. Standby is the least invasive OS energy saving state. Standby implies only that the PE is unavailable and does not clear any debug settings. For standby, the Debug architecture requires only the following: • If the external debug interface is accessed, the PE must respond to that access. Arm recommends that, if the PE executed a <code>WFI</code> or <code>WFE</code> instruction to enter standby, then it does not retire that instruction. Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.
Retention	The PE state, including debug settings, is preserved in low-power structures, allowing the Core power domain to be at least partially turned off. Changing from low-power retention to normal operation does not involve a reset of the PE. The saved PE state is restored on changing from low-power retention state to normal operation. If software has to use an IMPLEMENTATION DEFINED code sequence before entering, or after leaving, a retention state, this is referred to as a <i>software-visible retention state</i>. It is IMPLEMENTATION DEFINED whether the value of <code>DBGPRCR.CORENPDRQ</code> is set to its Cold reset value on leaving the software-visible retention state. See the description of <code>DBGPRCR.CORENPDRQ</code> for more information. Note • This model of retention does not include implementations where the PE exits the state in response to a debug register access. From the Debug architecture perspective, implementations like this are forms of standby.
Powerdown	These measures must include the OS saving any PE state, including the debug settings, that must be preserved over powerdown. If <code>FEAT_DoubleLock</code> is implemented, it is used during powerdown. Changing from powerdown to normal operation must include: • A Cold reset of the PE after the power level has been restored. • The OS restoring the saved PE state. External Debug Request debug events stay pending and debug registers in the Core power domain cannot be accessed.

An implementation might support enabling and disabling threads, either dynamically or once at reset time. Threads that are disabled in this way must appear to the external debugger as either:

- Powered off, meaning they are either:
 - In a powerdown state.

- In a retention state.
- Held in reset state.

Arm A-profile external debug uses a simpler two states model for the Debug power domain. The two states are:

Off The Debug power domain is turned off.

On The Debug power domain is turned on.

The available power states, including the cross-product of Core power domain and Debug power domain power states is IMPLEMENTATION DEFINED. Implementations are not required to implement all of these states and might include additional states. These additional states must appear to the debugger as one of the logical power states defined by this model. The control of power states is IMPLEMENTATION DEFINED.

———— **Note** —————

As a result, it is IMPLEMENTATION DEFINED whether it is possible for the Debug power domain to be on when the Core power domain is off.

If the Debug power domain is implemented but is not powered up when the Core power domain is powered up, the Reset Catch debug event and the OS Unlock Catch debug event are disabled.

E6.4 Powerup request mechanism

If a powerup request mechanism is implemented, asserting the powerup request requests the power controller to power up the Core power domain, and to emulate any subsequent powerdown requests, until the powerup request mechanism is deasserted.

E6.4.1 Powerup request mechanism

[EDPRCR.COREPURQ](#) is the powerup request mechanism.

The control registers [DBGPRCR.CORENPDRQ](#) and [EDPRCR.COREPURQ](#) provide an interface between the power controller and the PE. They typically map directly to signals in the recommended external debug interface.

On Cold reset, if FEAT_DoPD is not implemented, [DBGPRCR.CORENPDRQ](#) is set to the value of [EDPRCR.COREPURQ](#).

E6.5 Emulating low-power states

[DBGPRCR](#).CORENPDRQ and the powerup request mechanism can request the power controller to emulate states where the Core power domain is completely off or in a low-power state where the Core power domain registers cannot be accessed. This simplifies the requirements on software by sacrificing entirely realistic behavior.

If FEAT_DoPD is not implemented, [EDPRSR](#).{SPD, PU} indicates the Core power domain power state. For more information, see:

- The [DBGPRCR_EL1](#) and [DBGPRCR](#) System register descriptions.
- The [EDPRCR](#) and [EDPRSR](#) external debug register descriptions.
- Recommended External Debug Interface.

The measures to emulate powerdown are IMPLEMENTATION DEFINED. The ability of the debugger to access the state of the PE and the system might be limited as a result of the measures adopted.

In an emulated powerdown state, the debugger must be able to access all debug, PMU, CTI, and trace unit registers that are accessible on the external debug interface and are in one of:

- The Debug power domain.
- The Core power domain.
- When a trace unit with a separate trace unit Core power domain is implemented, and the trace unit Core power domain is powered on, the trace unit Core power domain.

That is, the debugger must be able to read and write to such registers without receiving errors. This allows an external debugger to debug the powerup sequence.

Arm recommends that any IMPLEMENTATION DEFINED registers that are on the external debug interface and in either the Core power domain or the Debug power domain are also accessible in an emulated powerdown state.

If [FEAT_DoubleLock](#) is implemented, `DoubleLockStatus() == FALSE` when [DBGPRCR](#).CORENPDRQ == 1.

Otherwise, the behavior of the PE in emulated powerdown must be similar to that in a real powerdown state. In particular, the PE must not respond to other system stimuli, such as interrupts.

[Example E6-1](#) and [Example E6-2](#) are examples of two approaches to emulating powerdown.

Example E6-1 An example of emulating powerdown

The PE is held in Standby state, isolated from any system stimuli. It is IMPLEMENTATION DEFINED whether the PE can respond to debug stimuli such as an External Debug Request debug event.

If the PE can enter Debug state, then the external debugger is able to use the ITR to execute instructions, such as loads and stores. This causes the external debugger to interact with the system. If the external debugger restarts the PE, the PE leaves Standby state and restarts fetching instructions from memory.

Example E6-2 Another example of emulating powerdown

The PE is held in Warm reset. This limits the ability of an external debugger to access the resources of the PE. For example, the PE cannot be put into Debug state.

On exit from emulated powerdown the PE is reset. However, the debug registers that are only reset by a Cold reset must not be reset. Typically this means that a Warm reset is substituted for the Cold reset. As such, the effect of accessing any register that is reset by a Warm reset while the PE is in the emulated powerdown state will have an IMPLEMENTATION DEFINED effect on that register.

Note

- Warm reset and Cold reset have different effects apart from resetting the debug registers. In particular, RMR_ELx is reset by a Cold reset and controls the reset state on a Warm reset. This means that if a Cold reset is substituted by a Warm reset, the behavior of the reset code might be different.
 - The timing effects of powering down are typically not factored in the powerdown emulation. Examples of these timing effects are clock and voltage stabilization.
 - Emulation does not model the state lost during powerdown, meaning that it might mask errors in the state storage and recovery routines.
-

E6.6 Debug OS Save and Restore sequences

From the introduction of , the following registers provide the OS Save and Restore mechanism:

- The *OS Lock Access Register*, [DBGOSLSR](#), locks the OS Lock to restrict access to debug registers before starting an OS Save sequence, and unlocks the OS Lock after an OS Restore sequence.
- The *OS Lock Status Register*, [DBGOSLSR](#), shows the status of the OS Lock.
- The PE can be configured to generate an [OS Unlock Catch debug event](#) when the OS Lock is unlocked.
- If [FEAT_DoubleLock](#) is implemented, the OS Double Lock locks out an external debug interface entirely. This is only used immediately before a powerdown sequence.

See also:

- [FEAT_DoubleLock](#)
- [Reset and debug](#)
- Example OS Save and Restore sequences

E6.6.1 EDPRSR.{DLK, SPD, PU} and the Core power domain

If FEAT_DoPD is not implemented, a debugger uses [EDPRSR](#).{DLK, SPD, PU} to determine whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read.

Table E6-1 Interpretation of the EDPRSR.{DLK, SPD, PU} bits

EDPRSR			Core power domain			Notes
DLK	SPD	PU	Power	Accesses	State lost	
0	0	1	On	OK	No	-
0	1	1	On	OK	Yes	SPD is cleared to 0 following the read.
1	X	1	On	Error	Not known	FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE. Software locks the OS Double Lock before removing power.
X	1	0	Off	Error	Yes	A Cold reset will be asserted on exiting powerdown state, but not on exiting low-power retention state.
X	0	0	Not known	Error	Not known	

If FEAT_DoPD is implemented, accesses to [EDPRSR](#) return an error when the Core power domain is off or in a retention state, meaning successful reads of [EDPRSR](#) always return 1 for [EDPRSR](#).PU.

When FEAT_Debugv8p4 is implemented, and whenever [FEAT_DoubleLock](#) is not implemented, [EDPRSR](#).DLK is always 0.

If [FEAT_DoubleLock](#) is not implemented, [DoubleLockStatus\(\)](#) always returns FALSE.

If the Core power domain is powered up and [DoubleLockStatus\(\)](#) == TRUE, then:

- When FEAT_Debugv8p2 is not implemented, [EDPRSR](#).{DLK, SPD, PU} can read either {1, UNKNOWN, 1} or {UNKNOWN, 0, 0}.
- When FEAT_Debugv8p2 is implemented, and FEAT_Debugv8p4 is not implemented, [EDPRSR](#).{DLK, SPD, PU} can only read {UNKNOWN, 0, 0}.

E6.6.2 EDPRSR.SPD when the Core domain is in either retention or powerdown state

When the Core power domain is in either the retention or powerdown state, **EDPRSR.SPD** is not cleared following a read of **EDPRSR** and it is IMPLEMENTATION DEFINED whether:

- **EDPRSR.SPD** shows whether the state of the debug registers in the Core power domain has been lost since the last time that **EDPRSR** was read. This means that:
 - When the Core power domain is in the powerdown state, **EDPRSR.SPD** is RAO, this indicates that the state of the debug registers has been lost.
 - When the Core power domain is in the retention state, **EDPRSR.SPD** indicates whether the state of the debug registers was lost before the Core power domain entered retention state.
- **EDPRSR.SPD** is RAZ, and:
 - On leaving the powerdown state, **EDPRSR.SPD** is set to 1 which indicates that the state of the debug registers has been lost.
 - On leaving the retention state, **EDPRSR.SPD** reverts the value it had on entering the retention state.

E6.6.3 EDPRSR.{DLK, R} and reset state

If **FEAT_DoubleLock** is not implemented, **DoubleLockStatus()** always returns FALSE.

If **FEAT_DoubleLock** is implemented and enabled, the behavior of all registers and fields except **EDPRSR.DLK** is the same as their behavior if **FEAT_Debugv8p4** is not implemented.

EDPRSR.R is UNKNOWN when **DoubleLockStatus() == TRUE**. **OSDLR_EL1.DLK** is cleared to 0 by a reset. If the Core power domain is powered up and entered reset state with the OS Double Lock locked, it is CONstrained UNPREDICTABLE whether a read of **EDPRSR** while the PE is in reset state returns:

- **EDPRSR.{DLK, R, PU} == {1, UNKNOWN, 1}** indicating that the OS Double Lock is locked. This is not permitted from Armv8.2.
- **EDPRSR.{DLK, R, PU} == {0, 1, 1}** indicating that the PE is in reset state.
- **EDPRSR.{DLK, R, PU} == {UNKNOWN, UNKNOWN, 0}** indicating that the registers in the Core power domain cannot be accessed because the OS Double Lock is locked.

If the PE was powered up and the OS Double Lock was unlocked when the PE was reset, then **EDPRSR.{DLK, R, PU}** reads as {0, 1, 1} while the PE is in reset state.

On leaving reset state, **EDPRSR.{DLK, R}** reads as {0, 0}.

E6.6.4 Debug registers to save over powerdown

Table E6-2 shows the different requirements for self-hosted debug over powerdown and external debug over powerdown:

- The column labeled Self-hosted lists registers that software must preserve over powerdown so that it can support self-hosted debug over powerdown. This does not require use of the OS Save and Restore mechanism.
- The column labeled External lists registers that software must preserve over powerdown so that it can support external debug over powerdown. This requires use of the OS Save and Restore mechanism:
 - Some external debug registers are not normally accessible to software executing on the PE. Additional debug registers are provided that give software the required access to save and restore these external debug registers when **DBGOSLSR.OSLK** is locked. These registers include **DBGOSECCR**, **DBGDTRRExt**, and **DBGDTRTExt**.
- Some registers might only present in some implementations, or might not be accessible at all Exception levels or in Non-secure state. **DBGVCR32_EL2** and **SDER32_EL3** are only required to support AArch32.

Table E6-2 does not include registers for the OPTIONAL Trace and Performance Monitor extensions.

Table E6-2 Debug registers to save over powerdown

Register in AArch32 state	Self-hosted	External
DBGDSCRExt	Yes	Yes
DBGBVR<n>	Yes	Yes
DBGBCR<n>	Yes	Yes
DBGWVR<n>	Yes	Yes
DBGWCR<n>	Yes	Yes
DBGVCR	Yes	-
HDCR	Yes	-
DBGDCCINT	-	Yes
DBGCLAIMSET , DBGCLAIMCLR	-	Yes
DBGOSECCR	-	Yes
DBGDTRRXext DBGDTRTXext	-	Yes

E6.6.5 OS Save sequence

To preserve the debug logic state over a powerdown, the state must be saved to nonvolatile storage. This means the OS Save sequence must:

1. Lock the OS Lock by writing the key value `0xC5ACCE55` to the [DBGOSLAR](#) in AArch32 state.
2. Execute an `ISB` instruction.
3. Walk through the debug registers listed in [Debug registers to save over powerdown](#) and save the values to the nonvolatile storage.

If the [FEAT_DoubleLock](#) is implemented, before removing power from the Core power domain, software must:

1. Lock the OS Double Lock by writing 1 to [DBGOSDLR.DLK](#) in AArch32 state.
2. Execute a [Context Synchronization event](#).

E6.6.6 OS Restore sequence

After a powerdown, the OS Restore sequence must perform the following steps to restore the debug logic state from the non-volatile storage:

1. Lock the OS Lock, as described in [OS Save sequence](#). The OS Lock is generally locked by the Cold reset, but this step ensures that it is locked.
2. Execute an `ISB` instruction.
3. To ensure that, if an external debugger clears the OS Lock before the end of this sequence, no debug exceptions are generated:
 - Write 0 to `MDSCR_EL1` if executing in AArch64 state.
 - Write 0 to [DBGDSCRExt](#) if executing in AArch32 state.
4. Write 0 to [DBGDSCRExt](#) to ensure that, if an external debugger clears the OS Lock before the end of this sequence, no debug exceptions are generated.
5. Walk through the debug registers listed in [Debug registers to save over powerdown](#), and restore the values from the nonvolatile storage. The last register to be restored must be:
 - `MDSCR_EL1` if executing in AArch64 state.
 - [DBGDSCRExt](#) if executing in AArch32 state.

6. Walk through the debug registers listed in [Debug registers to save over powerdown](#), and restore the values from the nonvolatile storage. The last register to be restored must be [DBGDSCRext](#).
7. Execute an `ISB` instruction.
8. Unlock the OS Lock by writing 0 to [DBGOSLAR.OSLK](#).
9. Execute a [Context Synchronization event](#).

Note

The OS Restore sequence overwrites the debug registers with the values that were saved. If there are valid values in these registers immediately before the restore sequence, then those values are lost.

E6.6.7 Debug behavior when the OS Lock is locked

The main purpose of the OS Lock is to prevent updates to debug registers during an OS Save or OS Restore operation. The OS Lock is locked on a Cold reset.

When the OS Lock is locked:

- Access to debug System registers through the System register interface is mainly unchanged except that:
 - Certain registers are read and written without side-effects.
 - Fields in [DBGDSCRext](#) and [DBGOSECCR](#) that are normally read-only become read/write.This allows the state to be saved or restored. For more information, see the relevant register description in [External Debug Register Descriptions](#).
- Access to debug registers by the external debug interface is restricted to prevent an external debugger modifying the registers that are being saved or restored. For more information, see [External debug interface register access permissions summary](#).
- Debug exceptions, other than Breakpoint Instruction exceptions are not generated.
- Breakpoint and Watchpoint debug events are not generated. The OS Lock has no effect on Breakpoint Instruction exceptions and other debug events.

E6.6.8 Debug behavior when the OS Lock is unlocked

When the OS Lock is unlocked, the PE sets [EDES.OSUC](#) to 1 if the OS Unlock Catch debug event is enabled and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. See [OS Unlock Catch debug event](#).

E6.6.9 Debug behavior when the OS Double Lock is locked

If the [FEAT_DoubleLock](#) is implemented, software locks the OS Double Lock immediately before a powerdown sequence.

The OS Double Lock ensures that it is safe to remove core power by forcing the debug interfaces to be quiescent.

When [DoubleLockStatus\(\)](#) == TRUE:

- The external debug interface has only restricted access to the debug registers, so that it is quiescent before removing power. See [External debug interface register access permissions summary](#).
- Debug exceptions, other than Breakpoint Instruction exceptions, are not generated.
- Halting is prohibited. See [Halting allowed and halting prohibited](#).

Note

Pending Halting debug events might be lost when core power is removed.

- No asynchronous debug events are [WF*](#) wakeup events.

If the [FEAT_DoubleLock](#) is not implemented, the PE ensures these conditions are met before allowing power to be removed.

Software must synchronize the update to [DBGOSDLR](#) before it indicates to the system that core power can be removed. The interface between the PE and its power controller is IMPLEMENTATION DEFINED.

Typically software indicates that core power can be removed by entering the Wait For Interrupt state. This means that software must explicitly synchronize the [DBGOSDLR](#) update before issuing the [WFI](#) instruction.

[DBGOSDLR.DLK](#) is ignored and [DoubleLockStatus\(\)](#) == FALSE if either:

- The PE is in Debug state.
- [DBGPRCR.CORENPDRQ](#) is set to 1.

Note

It is possible to enter Debug state with [DBGOSDLR.DLK](#) set to 1. This is because a [Context Synchronization event](#) is required to ensure the OS Double Lock is locked, meaning that Debug state might be entered before the [DBGOSDLR](#) update is synchronized.

Because [DBGOSDLR.DLK](#) is ignored when [DBGPRCR.CORENPDRQ](#) is set to 1, an external debugger can write to [DBGPRCR.CORENPDRQ](#), and the [FEAT_DoubleLock](#) is not always implemented, software must not rely on using the OS Double Lock to disable debug exceptions or to prohibit halting, or both. Arm deprecates use of the OS Double Lock for these purposes, and instead recommends that software:

- Uses the OS Lock to disable debug exceptions during save or restore sequences.
- Uses the debug authentication interface to prohibit halting and external debug access to debug registers at times other than immediately prior to removing power.

As the purpose of the OS Double Lock is to ensure that it is safe to remove core power, if the [FEAT_DoubleLock](#) is implemented, it is important to avoid race conditions that defeat this purpose. Arm recommends that:

- Once the write to [DBGOSDLR.DLK](#) has been synchronized by a [Context Synchronization event](#) and [DoubleLockStatus\(\)](#) == TRUE, a PE must:
 - Not allow a debug event generated before the [Context Synchronization event](#) to cause an entry to Debug state or act as a wakeup event for a [WFI](#) or [WFE](#) instruction after the [Context Synchronization event](#) has completed.
 - Complete any external debug access started before the [Context Synchronization event](#) by the time the [Context Synchronization event](#) completes.

Note

A debug register access might be in progress when software sets [DBGOSDLR.DLK](#) to 1. An implementation must not permit the synchronization of locking the OS Double Lock to stall indefinitely while waiting for that access to complete. This means that any debug register access that is in progress when software sets [DBGOSDLR.DLK](#) to 1 must complete or return an error in finite time.

- If a write to [DBGPRCR](#) or [EDPRCR](#) made when [DBGOSDLR.DLK](#) == 1 changes [DBGPRCR.CORENPDRQ](#) or [EDPRCR.CORENPDRQ](#) from 1 to 0, meaning [DoubleLockStatus\(\)](#) changes from FALSE to TRUE, then before signaling to the system that the CORENPDRQ field has been cleared and emulation of powerdown is no longer requested, meaning the system can remove core power, the PE must ensure that all the requirements for [DoubleLockStatus\(\)](#) == TRUE listed in this section are met.

In a standard OS Save sequence, the OS Lock is locked before the OS Double Lock is locked. This means that writes to CORENPDRQ are ignored by the time the OS Double Lock is locked. However, if [DoubleLockStatus\(\)](#) == FALSE, an external debugger can clear the OS Lock at any time, and then write to [EDPRCR](#).

E6.7 Reset and debug

All registers in the Core power domain are either:

- Reset by both a Cold and a Warm reset.
- Reset only by a Cold reset and are not changed by a Warm reset.

For more information, see [Resets and power domains](#).

All registers in the Debug power domain are reset by an External Debug reset.

[Figure E6-1](#) shows this reset scheme. The following three reset signals are an example implementation of the reset scheme:

- **CORERESET**, which must be asserted for a Warm reset.
- **CPUPORESET**, which must be asserted for a Cold reset.
- **PRESETDBG**, which must be asserted for an External Debug reset.

As shown in the figure, the external debug logic is split between the Debug power domain and the Core power domain.

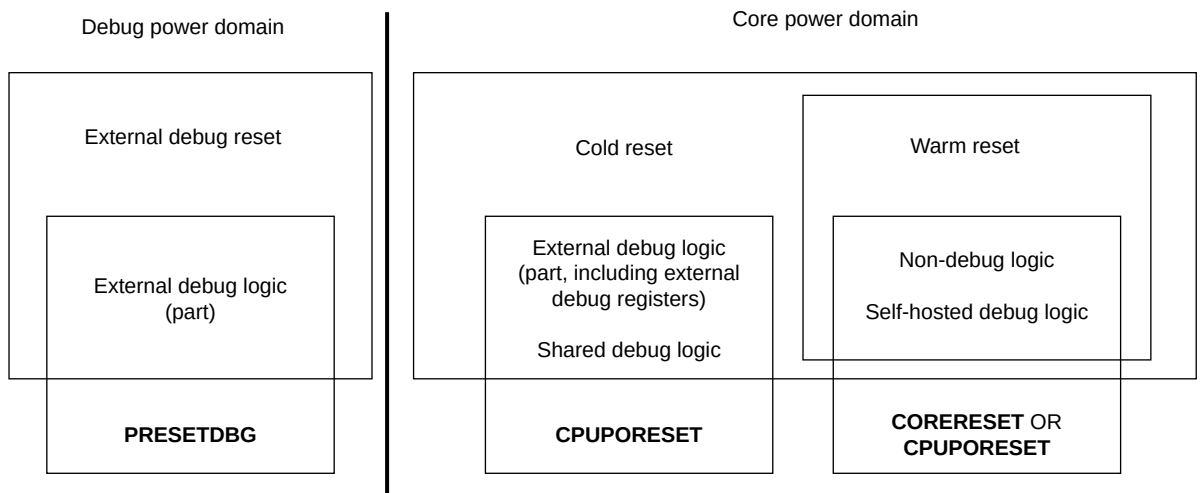


Figure E6-1 Power and reset domains

For more information about power domains and power states, see [Power domains and debug](#).

When power is first applied to the Debug power domain, **PRESETDBG** must be asserted.

When power is first applied to the Core power domain, **CPUPORESET** must be asserted.

Note

In this scheme, logic in the Warm reset domain is reset by asserting either **CORERESET** or **CPUPORESET**. This implies a particular implementation style that permits these approaches.

CPUPORESET is not normally asserted on moving from a low-power state, where power has not been removed, to a full-power state. This can occur, for example, on exiting a low-power retention state. See also [Emulating low-power states](#) and the [EDPRSR](#) register description.

E6.7.1 External debug interface accesses to registers in reset

If a reset signal is asserted and the external debug interface:

- Writes a register, or indirectly writes a register or register field as a side-effect of an access:

- Then, if the register or register field is reset by that reset signal, it is **CONSTRAINED UNPREDICTABLE** whether the register or register field takes the reset value or the value written. The reset value might be **UNKNOWN**.
- Otherwise, the register or register field takes the value that is written.
- Reads a register, or indirectly reads a register or register field, as part of an access:
 - Then, if the register or register field is reset by that reset signal, the value returned is **UNKNOWN**.
 - Otherwise, the value of the register or register field is returned.

It is **IMPLEMENTATION DEFINED** whether any register can be accessed when External Debug reset is being asserted. The result of these accesses is **IMPLEMENTATION DEFINED**.

Chapter E7

The PC Sample-based Profiling Extension

This chapter describes the OPTIONAL PC Sample-based Profiling Extension that provides a non-invasive external debug component.

It contains the following section:

- [About the PC Sample-based Profiling Extension](#)

E7.1 About the PC Sample-based Profiling Extension

The PC Sample-based Profiling Extension is an **OPTIONAL** extension that provides coarse-grained, non-invasive profiling by an external debugger. See also [Non-invasive behavior](#).

When the PC Sample-based Profiling Extension is implemented, the PE implements a PC sample register and one or more context sample registers as external registers. Reads of the PC sample register return a PC sample value and capture the PE context in the context sample registers. For example, tools can use these samples to populate a statistical model of the performance of software executing on the PE.

Note

Data returned by periodic sampling of PC Sample-based Profiling registers is sufficient to allow tools to estimate the distribution of time spent executing software on the PE.

The delay between an instruction being executed by the PE and its address appearing in the PC Sample Register is not defined, and the architecture does not require that the sampled instruction was recently executed. For example, if a piece of software executes a load instruction that reads the PC Sample Register of the PE it is running on, there is no guaranteed relationship between the address of the load instruction and the value read. The PC Sample Register is intended only for use by an external agent to provide statistical information for software profiling.

It must be possible to sample references to branch targets. It is **IMPLEMENTATION DEFINED** whether references to other instructions can be sampled. The branch target for a conditional branch instruction that fails its condition check is the instruction that follows the conditional branch instruction. The branch target for an exception is the exception vector address.

Note

In the Armv7 PC Sample-based Profiling Extension, an offset was applied to the sampled Program Counter value and this offset and the instruction set state indicated in bits [1:0] of the sampled value. From the introduction of the Armv8 PC Sample-based Profiling Extension, the sampled value is the address of an instruction that has executed, with no offset and no indication of the instruction set state.

- [Controlling the PC Sample-based Profiling Extension](#).
- [Registers implemented by the PC Sample-based Profiling Extension](#).
- [Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN](#).
- [Pseudocode description of PC Sample-based Profiling](#).

E7.1.1 Controlling the PC Sample-based Profiling Extension

PC Sample-based Profiling is controlled by the **IMPLEMENTATION DEFINED** authentication interface [ExternalNoninvasiveDebugEnabled\(\)](#).

PC Sample-based Profiling is allowed if both of the following apply:

The state of the **IMPLEMENTATION DEFINED** authentication interface is visible through `DBGAUTHSTATUS_EL1`. See Recommended authentication interface.

E7.1.1.1 Suspending and activating PC Sample-based Profiling

When all of the following apply, reading `PMPCSR[31:0]` or `PMPCSR[63:0]` makes PC Sample-based Profiling active:

- PC Sample-based Profiling is suspended.
- PC sampling is allowed.

A subsequent external read of `PMPCSR` will observe the updated state.

E7.1.2 Registers implemented by the PC Sample-based Profiling Extension

The PC Sample-based Profiling Extension is implemented by either [FEAT_PCSRv8](#):

- If [FEAT_PCSRv8](#) is implemented then the PC Sample-based Profiling Extension registers are implemented in the external debug register space. [EDDEVID1](#).PCSample identifies when [FEAT_PCSRv8](#) is implemented and which PC Sample-based Profiling Extension registers are implemented.

If [FEAT_PCSRv8](#) is implemented:

- The following external debug registers can be implemented:
 - [EDCIDSR](#).
 - [EDPCSR](#).
 - [EDVIDSR](#).

See [External debug interface registers](#).

E7.1.3 Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN

The architecture permits IMPLEMENTATION DEFINED extensions to external debug to define mechanisms that make the values of the PC Sample-based profiling registers UNKNOWN. However, it requires that any such mechanism is disabled by default. This means that powerup or a hard reset of the PE must leave the PE in a state where the PC Sample-based Profiling Extension, if implemented, exhibits its architecturally-defined behavior.

Note

A mechanism that, when enabled, makes the PC Sample-based profiling registers UNKNOWN might use other sample-based profiling events that are appropriate for a use that is independent of PC Sample-based Profiling.

If no branch instruction has been retired since the PE left Debug state, suspended state, or a state where PC Sample-based profiling is prohibited, then the sampled value is UNKNOWN.

If all of the following apply, then a read of PMPCSR[31:0] or PMPCSR[63:0] is permitted but not required to return 0xFFFF_FFFF for PMPCSR[31:0].

- The PE has not retired any branch instruction since leaving a state where PC Sample-based Profiling was suspended.
- Sampling on reads is not disabled.

This includes a read of PMPCSR[31:0] or PMPCSR[63:0] that indirectly makes PC Sample-based Profiling active.

E7.1.4 Pseudocode description of PC Sample-based Profiling

When [FEAT_PCSRv8](#) is implemented, the functionality is described by the pseudocode functions:

- [CreatePCSample\(\)](#), which populates a variable of type [PCSample](#).
- [Read_EDPCSRlo\(\)](#), which writes a PC sample to the [EDPCSR](#) and associated registers.

Chapter E8

About the External Debug Registers

This chapter provides some additional information about the external debug registers. It contains the following sections:

- [Relationship between external debug and System registers](#)
- [Endianness and supported access sizes](#)
- [Synchronization of changes to the external debug registers](#)
- [Memory-mapped accesses to the external debug interface](#)
- [External debug interface register access permissions](#)
- [External debug interface registers](#)
- [Cross-trigger interface registers](#)
- [External debug register resets](#)

E8.1 Relationship between external debug and System registers

Table E8-1 shows the relationship between external debug registers and System registers. Where no relationship exists, the registers are not listed.

Table E8-1 Equivalence between external debug and System registers

External debug register	System register	
	AArch32	Notes
DBGDTRRX_EL0	DBGDTRRXint	See also Summary of System register accesses to the DCC
DBGDTRTX_EL0	DBGDTRTXint	
OSLAR_EL1	DBGOSLAR	-
DBGBVR<n>_EL1[31:0] DBGBVR<n>_EL1[63:32]	DBGBVR<n> DBGBXVR<n>	-
DBGBCR<n>_EL1[31:0]	DBGBCR<n>	-
DBGWVR<n>_EL1[31:0]	DBGWVR<n>	-
DBGWCR<n>_EL1[31:0]	DBGWCR<n>	-
DBGCLAIMSET_EL1	DBGCLAIMSET	-
DBGCLAIMCLR_EL1	DBGCLAIMCLR	-
DBGAUTHSTATUS_EL1	DBGAUTHSTATUS	Read-only
EDSCR	DBGDSCRext	Only some fields map
EDECCR	DBGOSECCR	Applies when the OS Lock is locked
MIDR_EL1	MIDR	Read-only copies of Processor ID Registers
EDDEVAFF0	MPIDR	Read-only copies of system ID registers

In addition:

- [EDSCR](#).{TXfull, RXfull} are read-only aliases for [DBGDSCRint](#).{TXfull, RXfull}.
- [EDPRCR](#).CORENPDRQ is a read/write alias for [DBGPRCR](#).CORENPDRQ.
- [EDPRSR](#).OSLK is a read-only alias for [DBGOSLSR](#).OSLK.
- If the [FEAT_DoubleLock](#) is implemented, [EDPRSR](#).DLK is a read-only function of [DBGOSDLR](#).DLK.

E8.2 Endianness and supported access sizes

The debug registers, Performance Monitors registers, and CTI registers are implemented as memory-mapped peripherals. The Arm architecture requires memory-mapped peripherals to be little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the debug registers, Performance Monitors registers, and CTI registers, implementations must:

- Comply with the requirements of [Supported access sizes and Endianness](#).
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register, even if all components with direct memory access to the memory-mapped peripheral support making 64-bit accesses.

Note

These requirements mean that a system implementing the debug registers using a 32-bit bus, such as an AMBA APB3, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

For accesses from the external debug interface, the size of an access is determined by the interface. For an access from an ADiv5-compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register.

E8.3 Synchronization of changes to the external debug registers

This section describes the synchronization requirements for the external debug interface.

For more information on how these requirements affect debug, see:

- [Synchronization and debug exceptions](#) for exceptions taken from AArch32 state.
- [Synchronization and Halting debug events](#).
- [Synchronization of DCC and ITR accesses](#).

This section refers to accesses from the external debug interface as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

[Synchronization of changes to AArch32 System registers](#) define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads, and external writes as indirect writes.

For general information about synchronization, access completion, ordering, and observability, see [Synchronization of memory-mapped registers](#).

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. With the exception of [DBGBCR<n>_EL1](#), [DBGBVR<n>_EL1](#), [DBGWCR<n>_EL1](#), and [DBGWVR<n>_EL1](#), external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

[Synchronization of DCC and ITR accesses](#) describes the synchronization requirements for the DCC and ITR.

Changes to the IMPLEMENTATION DEFINED authentication interface are external writes to the authentication status registers by the Requester of the authentication interface. See [Synchronization and the authentication interface](#).

The external agent must be able to guarantee completion of a write. For example by:

- Marking the memory as Device-nGnRnE and executing a DSB barrier, if the system supports this property.
- Reading back the value written.
- Some guaranteed property of the connection between the PE and the external agent.

Note

For an external Debug Access Port, access completion is an IMPLEMENTATION DEFINED property. For a CoreSight system using APB-AP to access a debug APB, accesses complete in order.

However, the external agent cannot force synchronization of completed writes without halting the PE. Executing an ISB instruction, either in Debug state or in Non-debug state, and exiting from Debug state forces synchronization. If the PE is in Debug state, executing an ISB instruction is guaranteed to explicitly synchronize any external reads, external writes, and changes to the authentication interface that are ordered before the external write to [EDITR](#).

For any given observer, external writes to the following register groups are guaranteed to be observable in the same order in which they complete:

- The breakpoint registers, [DBGBCR<n>_EL1](#) and [DBGBVR<n>_EL1](#).
- The watchpoint registers, [DBGWCR<n>_EL1](#) and [DBGWVR<n>_EL1](#).

This guarantee applies only to external writes to registers within one of these groups. There is no guarantee regarding the ordering of the observability of external writes within these groups with respect to external writes to registers, for example [EDSCR](#), or between breakpoints and watchpoints, including watchpoints linked to context matching breakpoints.

Note

This means that a debugger can rely on the external writes to be observed in the same order in which they complete. It does not mean that a debugger can rely on the external writes being observed in finite time.

In a simple sequential execution an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

If two or more interfaces simultaneously access a register, the behavior must be as if the accesses occurred atomically and in any order. This is described in [Examples of the synchronization of changes to the external debug registers](#).

Note

Where simultaneous writes occur, once an order has been established by the implementation, some registers have behavior that is incompatible with the implementation simply dropping the older write and only performing the actions of the younger write. In these cases, the implementation must ensure that the register update is consistent with both writes having occurred. For example, writing one to write-one-to-clear or write-one-to-set fields in the older write can produce effects that persist even after writing zero to the same field in the younger write. Examples of registers with this behavior include [DBGCLAIMSET_EL1](#), [DBGCLAIMCLR_EL1](#), [PMCR_EL0](#), [PMOVSSET_EL0](#), [PMOVSCCLR_EL0](#), [PMCNTENSET_EL0](#), [PMCNTENCLR_EL0](#), [PMINTENSET_EL1](#), [PMINTENCLR_EL1](#), [PMSWINC_EL0](#), and [PMZR_EL0](#), all of which are simultaneously accessible through external debug and System register access.

E8.3.1 Synchronization and the authentication interface

Changes to the authentication interface are indirect writes to the state of the PE by the Requester of the authentication interface.

For an external debug interface read of any Authentication Status register, or an indirect read of the authentication interface made in determining the response to a subsequent external debug interface access, a change on the authentication interface must be observable following a subsequent explicit [Context Synchronization event](#), and:

- It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable in finite time.
- It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable following an entry to Debug state.

For a System register read of [DBGAUTHSTATUS_EL1](#), a change on the authentication interface is guaranteed to be observable only after a [Context Synchronization event](#).

Note

- In some systems, the authentication interface is fixed by configuration or is changed under the control of software. These systems can require explicit synchronization for any change to the authentication interface.
 - In other systems, the authentication interface is controlled dynamically by an external agent. In these systems, it is desirable that changes to the authentication interface do not require explicit synchronization by software executing on the PE to be observable by subsequent external debug interface accesses, and are either observable in finite time or are synchronized by entry to Debug state. Otherwise, there are scenarios where a debugger is not able to halt and debug the system.
-

E8.3.2 Examples of the synchronization of changes to the external debug registers

[Example E8-1](#), [Example E8-2](#), and [Example E8-3](#) show the synchronization of changes to the external debug registers.

Example E8-1 Order of synchronization of Breakpoint and Watchpoint register writes

Initially `DBGBVR<n>_EL1` is `0x8000` and `DBGBCR<n>_EL1` is `0x0181`. This means that a breakpoint is enabled on the halfword T32 instruction at address `0x8000`.

A sequence of external writes occurs in the following order:

1. `0x0000` is written to `DBGBCR<n>_EL1`, disabling the breakpoint.
2. `0x9000` is written to `DBGBVR<n>_EL1[31:0]`.
3. `0x0061` is written to `DBGBCR<n>_EL1`, enabling a breakpoint on the halfword at address `0x9002`.

The external writes must be observable to indirect reads in the same order as the external writes complete. This means that at no point is there a breakpoint enabled on either of the halfwords at address `0x8002` and `0x9000`.

Similarly a breakpoint or watchpoint must be disabled:

- If both halves of a 64-bit address have to be updated.
- If any of the `DBGBCR<n>_EL1` or `DBGWCR<n>_EL1` fields are modified at the same time as updating the address.

Example E8-2 Simultaneous accesses to DTR registers

Initially `EDSCR.{TXfull, TXU, ERR}` are 0. Then:

- `0x0DCCDA7A` is directly written to `DBGDTRTX_EL0` by an MSR instruction.
- `DBGDTRTX_EL0` is indirectly read by the external debug interface.

These accesses might happen at the same time and in any order.

If the direct write of `0x0DCCDA7A` to `DBGDTRTX_EL0` is handled first, then:

- The external debug interface read of `DBGDTRTX_EL0` clears `EDSCR.TXfull` to 0.
- `EDSCR.{TXU, ERR}` are unchanged.
- The external debug interface read returns `0x0DCCDA7A`.

If the indirect read of `DBGDTRTX_EL0` by the external debug interface is handled first, then:

- The external debug interface read of `DBGDTRTX_EL0` causes an underrun and as a result `EDSCR.{TXU, ERR}` are both set to 1.
- The external debug interface returns an UNKNOWN value.
- Writing `0x0DCCDA7A` to `DBGDTRTX_EL0` sets `DTRTX` to `0x0DCCDA7A` and `EDSCR.TXfull` to 1.

Example E8-3 Simultaneous writes to CLAIM registers

Initially all CLAIM tag bits are 0. Then:

- 0x01 is written to `DBGCLAIMSET_EL1` by a direct write, followed by an explicit [Context Synchronization event](#).
- 0x02 is written to `DBGCLAIMSET_EL1` by an external write.

These events might happen at the same time and in either order.

After this:

- `DBGCLAIMCLR_EL1` is read by a direct read.
- `DBGCLAIMCLR_EL1` is read by an external read.

In this case, a direct read can return either 0x01 or 0x03, and the external read can return either 0x02 or 0x03.

The only permitted final result for the CLAIM tags is the value 0x03, because this would be the result regardless of whether 0x01 or 0x02 is written first. This is because the external write is guaranteed to be observable to a direct read in finite time. See Synchronization requirements for AArch64 System registers.

It is not possible for a direct read to return 0x01 and the external read to return 0x02, because the writes to `DBGCLAIMCLR_EL1` are serialized.

In the following scenario, there is only one permitted result. Both observers observe the value 0x03, and then, at the same time, two writes occur:

- 0x04 is written to `DBGCLAIMSET_EL1` by a direct write, followed by an explicit [Context Synchronization event](#).
- 0x01 is written to `DBGCLAIMCLR_EL1` by an external write.

In this case, the only permitted final result for the CLAIM tags is the value 0x06.

E8.4 Memory-mapped accesses to the external debug interface

Support for memory-mapped access to the external debug interface is **OPTIONAL**. When memory-mapped access to the external debug interface is supported, the external debug interface is accessed as a little-endian memory-mapped peripheral.

If the external debug interface is CoreSight compliant, then an **OPTIONAL** Software Lock can be implemented for memory-mapped accesses to each component.

The Software Lock is **OPTIONAL** and deprecated. If it is not implemented, the behavior is as if it is unlocked. The Software Locks are controlled by [EDLSR](#) and [EDLAR](#), [PMLSR](#) and [PMLAR](#), and [CTILSR](#) and [CTILAR](#). See Management registers and CoreSight compliance.

With the exception of these registers and the effect of the Software Lock, the behavior of the memory-mapped accesses is the same as for other accesses to the external debug interface.

Note

The recommended memory-mapped accesses to the external debug interface are not compatible with the memory-mapped interface defined in Armv7. In particular:

- The memory map is different.
 - Memory-mapped accesses do not behave differently to Debug Access Port accesses when [DBGOSLSR.OSLK](#) == 1, meaning that the OS Lock is locked.
-

The following sections give more information about these memory-mapped accesses:

- [Register access permissions for memory-mapped accesses.](#)
- [Synchronization of memory-mapped accesses to external debug registers.](#)

See also [Supported access sizes and Endianness](#).

E8.4.1 Register access permissions for memory-mapped accesses

It is **IMPLEMENTATION DEFINED** whether unprivileged memory-mapped accesses are allowed. Privileged software is responsible for controlling memory-mapped accesses using the MPU.

If memory-mapped accesses are made through an ADiv5 interface, the Debug Access Port can block the access using [DBGSWENABLE](#). This is outside the scope of the Arm architecture. See *Arm® Debug Interface Architecture Specification ADiv5.0 to ADiv5.2*.

E8.4.1.1 Effect of the **OPTIONAL** Software Lock on memory-mapped access

For memory-mapped accesses, if other controls permit access to a register, the **OPTIONAL** Software Lock is implemented, and [EDLSR.SLK](#), [PMLSR.SLK](#), or [CTILSR.SLK](#) is set to 1, meaning the Software Lock is locked, then with the exception of the LAR itself:

- If other controls permit access to a register, then writes are ignored. That is:
 - Read/write (RW) registers become read-only, writes ignored (RO/WI).
 - Write-only (WO) registers become writes ignored (WI).
- Reads and writes have no side-effects. A side-effect is where a direct read or a direct write of a register creates an indirect write of the same or another register. When the Software Lock is locked, the indirect write does not occur.
- Writes to [EDLAR](#), [PMLAR](#), and [CTILAR](#) are unaffected.

This behavior must also apply to all **IMPLEMENTATION DEFINED** registers.

For example, if [EDLSR.SLK](#) is set to 1:

- [EDSCR.{TXfull, TXU, ERR}](#) are unchanged by a memory-mapped read from [DBGDTRTX_EL0](#).
- [EDSCR.{RXfull, RXO, ERR}](#) are unchanged by a memory-mapped write to [DBGDTRRX_EL0](#) that is ignored.

- [EDSCR](#).{ITE, ITO, ERR} are unchanged by a memory-mapped write to [EDITR](#) that is ignored.
- [DBGOSLSR](#).OSLK is unchanged by a memory-mapped write to [OSLAR_EL1](#) that is ignored.
- [EDPCSR](#)[63:32], [EDCIDSR](#), and [EDVIDSR](#) are unchanged by a memory-mapped read from [EDPCSR](#)[31:0].

Note

Updating [EDVIDSR](#), [EDCIDSR](#), and [EDPCSR](#)hi are side-effects of reading [EDPCSR](#)lo, such that these registers contain the matching context for [EDPCSR](#)lo. The process that updates [EDPCSR](#)lo with PC samples is not a side-effect of the access. Reads of [EDPCSR](#)lo made when the Software Lock is locked can be used to profile software.

- [PMPCSR](#)[63:32], [PMCID1SR](#)/[PMCID2SR](#), and [PMVIDSR](#) are unchanged by a memory-mapped read from [PMPCSR](#)[31:0].

Note

Updating [PMVIDSR](#), [PMCID1SR](#)/[PMCID2SR](#), and [PMPCSR](#)[31:0] are side-effects of reading [PMPCSR](#)[63:32], such that these registers contain the matching context for [PMPCSR](#)[63:32]. The process that updates [PMPCSR](#)[63:32] with PC samples is not a side-effect of the access. Reads of [PMPCSR](#)[63:32] made when the Software Lock is locked can be used to profile software.

- [EDPRSR](#).{SDR, SPMAD, SDAD, SR, SPD} are unchanged by a memory-mapped read from [EDPRSR](#).
- [EDPRSR](#).SDAD is not set if an error response is returned due to a memory-mapped read or write of any debug register as the result of the value of the EDAD field.
- The CLAIM tags are unchanged by memory-mapped writes to [DBGCLAIMSET_EL1](#) and [DBGCLAIMCLR_EL1](#) which are ignored.

Similarly, if [PMLSR](#).SLK is set to 1, then [EDPRSR](#).SPMAD is not set if an error response is returned to a memory-mapped read or write of any Performance Monitors register due to the value of the EPMAD field.

E8.4.1.2 Behavior of a not permitted memory-mapped access

Where the architecture requires that an external debug interface access generates an error response, a memory-mapped access must also generate an error response. However, it is IMPLEMENTATION DEFINED how the error response is handled, as this depends on the system.

Arm recommends that the error is returned as either:

- A synchronous external Data Abort.
- An SError exception.

E8.4.2 Synchronization of memory-mapped accesses to external debug registers

The synchronization requirements for memory-mapped accesses to the external debug interface is described in [Synchronization of changes to the external debug registers](#) and [Synchronization of memory-mapped registers](#).

The synchronization requirements between different routes to the external debug interface, that is, between Debug Access Port accesses and memory-mapped accesses are IMPLEMENTATION DEFINED.

E8.5 External debug interface register access permissions

Some external accesses to debug registers and Performance Monitor registers are not permitted and return an error response if:

- The Core power domain is powered down or is in low-power state where the registers cannot be accessed.
- `DBGOSLSR.OSLK == 1`. The OS Lock is locked.
- `FEAT_DoubleLock` is implemented and `DoubleLockStatus() == TRUE`. The OS Double Lock is locked.
- The access is disabled by either the authentication interface or secure monitor.

Not all registers are affected in all of these cases. For more information, see [External debug interface register access permissions summary](#).

E8.5.1 External debug over powerdown and locks

Accessing registers using the external debug interface is not possible when the Debug power domain is off. In this case, all accesses return an error.

External accesses to debug and Performance Monitors registers in the Core power domain are not permitted and return an error response if:

- The Core power domain is off or in low-power state where the registers cannot be accessed.
- `DBGOSLSR.OSLK == 1`, meaning that the OS Lock is locked. This allows software to prevent external debugger modification of the registers while it saves and restores them over powerdown.
- `FEAT_DoubleLock` is implemented and `DoubleLockStatus() == TRUE`. This means that the OS Double Lock is locked. The OS Double Lock ensures that it is safe to remove Core power by forcing the debug interface to be quiescent.

If `FEAT_DoubleLock` is not implemented, the hardware must provide another method to safely remove Core power.

The OS Lock condition does not apply to the following debug registers:

- `OSLAR_EL1`. This means that an external debugger can override this lock.
- `EDESR`. This means that an external debugger can program a debug event for when software unlocks the OS Lock. See [OS Unlock Catch debug event](#).
- The ID registers that describe the PE to the debugger.

See also [Debug registers to save over powerdown](#).

E8.5.2 External access disabled

Accesses are further controlled by the external authentication interface. An untrusted external debugger cannot program the breakpoint and watchpoint registers to generate spurious debug exceptions. If external invasive debugging is not enabled, these external accesses to the registers are disabled.

The disable applies to the `DBGBVR<n>_EL1`, `DBGBCR<n>_EL1`, `DBGWVR<n>_EL1`, and `DBGWCR<n>_EL1` registers.

The external debug interface cannot access these registers if `ExternalInvasiveDebugEnabled() == FALSE`.

PEs might also provide an OPTIONAL external debug interface to the Performance Monitor registers.

The `AllowExternalPMUAccess()` pseudocode function describes these accessibility rules.

E8.5.3 Behavior of a not permitted access

For an external debug interface access by a Debug Access Port, the Debug Access Port receives the error response and must signal this to the external debugger. For an ADIV5 implementation of a Debug Access Port, the error sets a sticky error flag in the Debug Access Port that the debugger can poll, and that suppresses further accesses until it is explicitly cleared.

When an error is returned because external access is disabled, and this is the highest priority error condition, a sticky error flag in [EDPRSR](#) is indirectly written to 1 as a side-effect of the access:

- For a debug register access when `AllowExternalDebugAccess()` == FALSE, [EDPRSR.SDAD](#) is indirectly written to 1.
- For Performance Monitor register access when `AllowExternalPMUAccess()` == FALSE, [EDPRSR.SPMAD](#) is indirectly written to 1.

The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see [Register access permissions for memory-mapped accesses](#).

If no error is returned, or the error is returned because of a higher priority error condition, the flag in [EDPRSR](#) is unchanged.

See also [Behavior of a not permitted memory-mapped access](#).

For more information, see *Arm® Debug Interface Architecture Specification*.

E8.5.4 External debug interface register access permissions summary

For accesses to:

- IMPLEMENTATION DEFINED registers, see [IMPLEMENTATION DEFINED registers](#).
- OPTIONAL registers for CoreSight compliance, see .
- Reserved, unallocated, or unimplemented registers, writes to read-only registers, and reads of write-only registers, see [Reserved and unallocated registers](#).

For all other external debug interface, CTI, and Performance Monitor registers, [Table E8-3](#), [Table E8-4](#), [Table E8-6](#) and [Table F3-1](#), show the response of the PE to accesses by the external debug interface.

E8.5.5 IMPLEMENTATION DEFINED registers

Register access permissions for IMPLEMENTATION DEFINED debug, Performance Monitors, and CTI registers are IMPLEMENTATION DEFINED, other than when the additional constraints in [Register access permissions for memory-mapped accesses](#) apply to IMPLEMENTATION DEFINED registers.

E8.5.6 Reserved and unallocated registers

The default access requirements for reserved and unallocated registers are described in [Access requirements for reserved and unallocated registers](#).

Note

Reads of WO and writes to RO refers to the default access permissions for a register. For example, when the SLK field is 1, meaning that the relevant registers become RO, a memory-mapped write to a RW register is ignored, and not treated as a reserved access.

The following reserved registers are RES0 in all conditions, other than when debug power is off:

- All reserved CTI registers.
- For the debug registers, and Performance Monitors registers, if the implementation is CoreSight architecture compliant, and either FEAT_DoPD is not implemented or the Core power domain is on, all reserved registers in the range `0xF00 - 0xFFC`. See Management register access permissions.

Otherwise, the architecture defines that:

1. If debug power is off, all register accesses, including reserved accesses, return an error.
2. For reserved debug registers and Performance Monitors registers, if FEAT_DoPD is implemented, and the Core power domain is off or in a low-power state, the response is an error. Otherwise, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0, when any of the following hold:

Off	The Core power domain is either completely off or in a low-power state in which the Core power domain registers cannot be accessed.
DLK	FEAT_DoubleLock is implemented and DoubleLockStatus () == TRUE. The OS Double Lock is locked.
OSLK	DBGOSLSR.OSLK == 1. The OS Lock is locked.

3. In addition, for reserved debug registers in the address ranges 0x400 - 0x4FC and 0x800 - 0x8FC, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

EDAD [AllowExternalDebugAccess](#) () == FALSE. External debug is disabled.

Note

See also [Behavior of a not permitted access](#).

4. In addition, for reserved Performance Monitors registers in the address ranges 0x000 - 0xEFC, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

EPMAD [AllowExternalPMUAccess](#) () == FALSE. External Performance Monitor access is disabled.

Note

See also [Behavior of a not permitted access](#).

5. For reads of WO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a write to the location to return an error.
6. For writes of RO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a read to the location to return an error.
7. For reads and writes of locations for features that are not implemented, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires an access to the location to return an error if the feature is implemented.

E8.6 External debug interface registers

The external debug interface register map is described by:

- [Summary of external debug registers.](#)
- [Cross-trigger interface registers.](#)
- [Performance Monitors external register views.](#)

Table E8-2 Summary of external debug registers

Offset	Name	Description	Access
0x020	EDES	External Debug Event Status Register	RW
0x024	EDECR	External Debug Execution Control Register	RW
0x030	EDWAR	External Debug Watchpoint Address Register	RO
0x080	DBGDTRRX_EL0	Debug Data Transfer Register, Receive	RW
0x084	EDITR	External Debug Instruction Transfer Register	WO
0x088	EDSCR	External Debug Status and Control Register	RW
0x08C	DBGDTRTX_EL0	Debug Data Transfer Register, Transmit	RW
0x090	EDRCR	External Debug Reserve Control Register	WO
0x094	EDACR	External Debug Auxiliary Control Register	RW
0x098	EDECCR	External Debug Exception Catch Control Register	RW
0x300	OSLAR_EL1	OS Lock Access Register	WO
0x310	EDPRCR	External Debug Power/Reset Control Register	RW
0x314	EDPRSR	External Debug Processor Status Register	RO
0x400 + (16 * n)	DBGBVR<n>_EL1	Debug Breakpoint Value Registers	RW
0x408 + (16 * n)	DBGBCR<n>_EL1	Debug Breakpoint Control Registers	RW
0x800 + (16 * n)	DBGWVR<n>_EL1	Debug Watchpoint Value Registers	RW
0x808 + (16 * n)	DBGWCR<n>_EL1	Debug Watchpoint Control Registers	RW
0xD00	MIDR_EL1	Main ID Register	RO
0xD20	EDPFR	External Debug Processor Feature Register	RO
0xD28	EDDFR	External Debug Feature Register	RO
0xD60	EDAA32PFR	External Debug Auxiliary Processor Feature Register	RO
0xF00	EDITCTRL	External Debug Integration mode Control register	RW
0xFA0	DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register	RW
0xFA4	DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register	RW
0xFA8	EDDEVAFF0	External Debug Device Affinity register 0	RO
0xFAC	EDDEVAFF1	External Debug Device Affinity register 1	RO
0xFB0	EDLAR	External Debug Lock Access Register	WO
0xFB4	EDLSR	External Debug Lock Status Register	RO
0xFB8	DBGAUTHSTATUS_EL1	Debug Authentication Status Register	RO
0xFBC	EDDEVARCH	External Debug Device Architecture Register	RO
0xFC0	EDDEVID2	External Debug Device ID register 2	RO

Offset	Name	Description	Access
0xFC4	EDDEVID1	External Debug Device ID Register 1	RO
0xFC8	EDDEVID	External Debug Device ID register 0	RO
0xFCC	EDDEVTYPE	External Debug Device Type register	RO
0xFD0	EDPIDR4	External Debug Peripheral Identification Register 4	RO
0xFE0	EDPIDR0	External Debug Peripheral Identification Register 0	RO
0xFE4	EDPIDR1	External Debug Peripheral Identification Register 1	RO
0xFE8	EDPIDR2	External Debug Peripheral Identification Register 2	RO
0xFEC	EDPIDR3	External Debug Peripheral Identification Register 3	RO
0xFF0	EDCIDR0	External Debug Component Identification Register 0	RO
0xFF4	EDCIDR1	External Debug Component Identification Register 1	RO
0xFF8	EDCIDR2	External Debug Component Identification Register 2	RO
0xFFC	EDCIDR3	External Debug Component Identification Register 3	RO

E8.6.1 Access permissions for the External debug interface registers

and show the access permissions for the external debug interface registers in an Arm A-profile Debug implementation. The terms are defined as follows:

Domain This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

Conditions This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.

The conditions are:

Off The Core power domain is completely off, or in low-power state. In these cases the Core power domain registers cannot be accessed [EDPRSR.PU](#) will read as 0.

———— Note ————

When the Core power domain is off, or in a low-power state, a debugger is permitted to access a debug register that is implemented in the external Debug power domain. When the Debug power domain is off, all accesses to the registers in the external Debug power domain return an error.

DLK [FEAT_DoubleLock](#) is implemented and [DoubleLockStatus\(\)](#) == TRUE. The OS Double Lock is locked..

OSLK [DBGOSLSR.OSLK](#) == 1. The OS Lock is locked.

EDAD [AllowExternalDebugAccess\(\)](#) == FALSE. External debug access is disabled for the access. See also [Behavior of a not permitted access](#).

EPMAD [AllowExternalPMUAccess\(\)](#) == FALSE. Access to the external Performance Monitors is disabled for the access. See also [Behavior of a not permitted access](#).

SLK The Software Lock is implemented and [SoftwareLockStatus\(\)](#) == TRUE. This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked for the access. See [Register access permissions for memory-mapped accesses](#).

Default This provides the default access permissions, if there are no conditions that prevent access to the register.

The access permissions are:

- This means that the default access permission applies. See the Default column, or the SLK column, if applicable.
- RO** This means that the register or field is read-only, and:
 - Unless the register description states otherwise, a RO field in an RW register ignores writes.
 - Where the `SLK` control makes a RW register RO, the register ignores writes.
- RW** This means that the register or field is read/write. Individual fields within the register might be RO or WO. See the relevant register description for details.
- RC** This means that a read of the register bit clears the field to 0.
- WO** This means that the register or field is write-only. Unless the register description states otherwise, a WO field in a RW register returns an UNKNOWN value on a read of the register.
- WI** This means that the register or field ignores writes.
- IMP DEF** This means that the access permissions are IMPLEMENTATION DEFINED.

`DBGBVR<n>_EL1` and `DBGWVR<n>_EL1`, are 64-bit registers mapped to pairs of 32-bit locations in the external debug interface. Doubleword accesses to these register are not guaranteed to be 64-bit single copy atomic. Software must take care before altering the values of `DBGBVR<n>_EL1`, `DBGWVR<n>_EL1`, `DBGBCR<n>_EL1`, and `DBGWCR<n>_EL1` to prevent UNPREDICTABLE behavior. See [Endianness and supported access sizes](#) for more information.

If OPTIONAL memory-mapped access to the external debug interface is supported, there might be additional constraints on memory-mapped accesses. See [Register access permissions for memory-mapped accesses](#).

For the reset values for the external debug interface registers, see [[@tbl:CIHCDIDH0](#)].

Table E8-3 Access permissions for the external debug interface registers if FEAT_DoPD is implemented

Offset	Register	Domain	Conditions (priority from left to right)			Default
			Off	OSLK	EDAD	
0x020	EDES	Core	Error	-	-	RW
0x024	EDECR	Core	Error	-	-	RW
0x030	EDWAR [31:0]	Core	Error	Error	-	RO
0x034	EDWAR [63:32]	Core	Error	Error	-	RO
0x080	DBGDTRRX_EL0	Core	Error	Error	-	RW
0x084	EDITR	Core	Error	Error	-	WO
0x088	EDSCR	Core	Error	Error	-	RW
0x08C	DBGDTRTX_EL0	Core	Error	Error	-	RW
0x090	EDRCR	Core	Error	Error	-	WO
0x094	EDACR	Core	Error	IMP DEF	-	RW
0x098	EDECCR	Core	Error	Error	-	RW
0x300	OSLAR_EL1	Core	Error	-	Error	WO
0x310	EDPRCR	Core	Error	-	-	RW

Offset	Register	Domain	Conditions (priority from left to right)			
			Off	OSLK	EDAD	Default
0x314	EDPRSR	Core	Error	-	-	RO
0x400+16*n	DBGBVR<n>_EL1[31:0] ^a	Core	Error	Error	Error	RW
0x404+16*n	DBGBVR<n>_EL1[63:32] ^a	Core	Error	Error	Error	RW
0x408+16*n	DBGBCR<n>_EL1[31:0] ^a	Core	Error	Error	Error	RW
0x40C+16*n	DBGBCR<n>_EL1[63:32] ^a	Core	Error	Error	Error	RW
0x800+16*n	DBGWVR<n>_EL1[31:0] ^a	Core	Error	Error	Error	RW
0x804+16*n	DBGWVR<n>_EL1[63:32] ^a	Core	Error	Error	Error	RW
0x808+16*n	DBGWCR<n>_EL1[31:0] ^a	Core	Error	Error	Error	RW
0x80C+16*n	DBGWCR<n>_EL1[63:32] ^a	Core	Error	Error	Error	RW
0xD20	EDPFR[31:0]	Core	Error	-	-	RO
0xD24	EDPFR[63:32]	Core	Error	-	-	RO
0xD28	EDDFR[31:0]	Core	Error	-	-	RO
0xD2C	EDDFR[63:32]	Core	Error	-	-	RO
0xD60	EDAA32PFR[31:0]	Core	Error	-	-	RO
0xD64	EDAA32PFR[63:32]	Core	Error	-	-	RO
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	-	RW
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	-	RW
0xFA8	EDDEVAFF0	Core	Error	-	-	RO
0xFAC	EDDEVAFF1	Core	Error	-	-	RO
0xFB8	DBGAUTHSTATUS_EL1	Core	Error	-	-	RO
0xFC0	EDDEVID2	Core	Error	-	-	RO
0xFC4	EDDEVID1	Core	Error	-	-	RO
0xFC8	EDDEVID	Core	Error	-	-	RO

a. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.

Table E8-4 Access permissions for the external debug interface registers if FEAT_DoPD is not implemented

Offset	Register	Domain	Conditions (priority from left to right)				
			Off	DLK	OSLK	Default	SLK
0x020	EDESR	Core	Error	Error	-	RW	RO
0x024	EDECR	Debug	-	-	-	RW	RO
0x028	EDSCR2	Core	Error	Error	Error	RW	RO
0x030	EDWAR[31:0]	Core	Error	Error	Error	RO	-
0x034	EDWAR[63:32]	Core	Error	Error	Error	RO	-
0x080	DBGDTRRX_EL0	Core	Error	Error	Error	RW	RO
0x084	EDITR	Core	Error	Error	Error	WO	WI
0x088	EDSCR	Core	Error	Error	Error	RW	RO
0x08C	DBGDTRTX_EL0	Core	Error	Error	Error	RW	RO
0x090	EDRCR	Core	Error	Error	Error	WO	WI

Offset	Register	Domain	Conditions (priority from left to right)				
			Off	DLK	OSLK	Default	SLK
0x094	EDACR	IMP DEF	IMP DEF	IMP DEF	IMP DEF	RW	RO
0x098	EDECCR	Core	Error	Error	Error	RW	RO
0x300	OSLAR_EL1	Core	Error	Error	-	WO	WI
0x310	EDPRCR	Core and Debug ^a	-	-	-	RW	RO
0x314	EDPRSR	Core and Debug ^a	-	-	-	RO	RO
0x400+16*n	DBGBVR<n>_EL1[31:0] ^b	Core	Error	Error	Error	RW	RO
0x404+16*n	DBGBVR<n>_EL1[63:32] ^b	Core	Error	Error	Error	RW	RO
0x408+16*n	DBGBCR<n>_EL1[31:0] ^b	Core	Error	Error	Error	RW	RO
0x40C+16*n	DBGBCR<n>_EL1[63:32] ^b	Core	Error	Error	Error	RW	RO
0x800+16*n	DBGWVR<n>_EL1[31:0] ^b	Core	Error	Error	Error	RW	RO
0x804+16*n	DBGWVR<n>_EL1[63:32] ^b	Core	Error	Error	Error	RW	RO
0x808+16*n	DBGWCR<n>_EL1[31:0] ^b	Core	Error	Error	Error	RW	RO
0x80C+16*n	DBGWCR<n>_EL1[63:32] ^b	Core	Error	Error	Error	RW	RO
0xD20	EDPFR[31:0]	IMP DEF	IMP DEF ^c	IMP DEF ^c	-	RO	RO
0xD24	EDPFR[63:32]	IMP DEF	IMP DEF ^c	IMP DEF ^c	-	RO	RO
0xD28	EDDFR[31:0]	IMP DEF	IMP DEF ^c	IMP DEF ^c	-	RO	RO
0xD2C	EDDFR[63:32]	IMP DEF	IMP DEF ^c	IMP DEF ^c	-	RO	RO
0xD60	EDAA32PFR[31:0]	IMP DEF	IMP DEF ^c	IMP DEF ^c	-	RO	RO
0xD64	EDAA32PFR[63:32]	IMP DEF	IMP DEF ^c	IMP DEF ^c	-	RO	RO
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	Error	RW	RO
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	Error	RW	RO
0xFA8	EDDEVAFF0	Debug	-	-	-	RO	RO
0xFAC	EDDEVAFF1	Debug	-	-	-	RO	RO
0xFB8	DBGAUTHSTATUS_EL1	Debug	-	-	-	RO	RO
0xFC0	EDDEVID2	Debug	-	-	-	RO	RO
0xFC4	EDDEVID1	Debug	-	-	-	RO	RO
0xFC8	EDDEVID	Debug	-	-	-	RO	RO

a. Some bits are in the Debug power domain and some bits are in the Core power domain. See register field descriptions for information.

b. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.

c. It is IMPLEMENTATION DEFINED whether an error is returned. See [External debug over powerdown and locks](#). If no error is returned, the access is permitted.

E8.7 Cross-trigger interface registers

The Embedded Cross-Trigger Interface, CTI, is located within its own block of the external debug memory map. There must be one such block for each PE.

If the CTI of a PE does not implement the [CTIDEVAFF0](#) or [CTIDEVAFF1](#) registers it must be located 64KB above the debug registers in the external debug interface.

[Table E8-5](#) shows the CTI register map.

Table E8-5 Summary of the Embedded Cross-Trigger Interface, CTI, registers

Offset	Name	Description	Access
0x000	CTICONTROL	CTI Control register	RW
0x010	CTIINTACK	CTI Output Trigger Acknowledge register	WO
0x014	CTIAPPSET	CTI Application Trigger Set register	RW
0x018	CTIAPPCLEAR	CTI Application Trigger Clear register	WO
0x01C	CTIAPPULSE	CTI Application Pulse register	WO
0x020 + (4 * n)	CTIINEN<n>	CTI Input Trigger to Output Channel Enable registers	RW
0x0A0 + (4 * n)	CTIOUTEN<n>	CTI Input Channel to Output Trigger Enable registers	RW
0x130	CTITRIGINSTATUS	CTI Trigger In Status register	RO
0x134	CTITRIGOUTSTATUS	CTI Trigger Out Status register	RO
0x138	CTICHINSTATUS	CTI Channel In Status register	RO
0x13C	CTICHOUTSTATUS	CTI Channel Out Status register	RO
0x140	CTIGATE	CTI Channel Gate Enable register	RW
0x144	ASICCTL	CTI External Multiplexer Control register	RO
0x150	CTIDEVCTL	CTI Device Control register	RW
0xF00	CTIITCTRL	CTI Integration mode Control register	RW
0xFA0	CTICLAIMSET	CTI CLAIM Tag Set register	RW
0xFA4	CTICLAIMCLR	CTI CLAIM Tag Clear register	RW
0xFA8	CTIDEVAFF0	CTI Device Affinity register 0	RO
0xFAC	CTIDEVAFF1	CTI Device Affinity register 1	RO
0xFB0	CTILAR	CTI Lock Access Register	WO
0xFB4	CTILSR	CTI Lock Status Register	RO
0xFB8	CTIAUTHSTATUS	CTI Authentication Status register	RO
0xFBC	CTIDEVARCH	CTI Device Architecture register	RO
0xFC0	CTIDEVID2	CTI Device ID register 2	RO
0xFC4	CTIDEVID1	CTI Device ID register 1	RO
0xFC8	CTIDEVID	CTI Device ID register 0	RO
0xFCC	CTIDEVTYPE	CTI Device Type register	RO
0xFD0	CTIPIDR4	CTI Peripheral Identification Register 4	RO
0xFE0	CTIPIDR0	CTI Peripheral Identification Register 0	RO
0xFE4	CTIPIDR1	CTI Peripheral Identification Register 1	RO
0xFE8	CTIPIDR2	CTI Peripheral Identification Register 2	RO

Offset	Name	Description	Access
0xFEC	CTIPIDR3	CTI Peripheral Identification Register 3	RO
0xFF0	CTICIDR0	CTI Component Identification Register 0	RO
0xFF4	CTICIDR1	CTI Component Identification Register 1	RO
0xFF8	CTICIDR2	CTI Component Identification Register 2	RO
0xFFC	CTICIDR3	CTI Component Identification Register 3	RO

[Table E8-6](#) shows the access permissions for the CTI registers in an Arm A-profile Debug implementation. For a definition of the terms used, see [External debug interface registers](#).

Table E8-6 Access permissions for the CTI registers

Offset	Register	Domain	Conditions (priority from left to right)				Default	SLK
			Off	DLK	OSLK	EDAD		
0x000	CTICONTROL	Debug	-	-	-	-	RW	RO
0x010	CTIINTACK	Debug	-	-	-	-	WO	WI
0x014	CTIAPPSET	Debug	-	-	-	-	RW	RO
0x018	CTIAPPCLEAR	Debug	-	-	-	-	WO	WI
0x01C	CTIAPPULSE	Debug	-	-	-	-	WO	WI
0x020+4×n	CTIINEN<n>	Debug	-	-	-	-	RW	RO
0x0A0+4×n	CTIOUTEN<n>	Debug	-	-	-	-	RW	RO
0x130	CTITRIGINSTATUS	Debug	-	-	-	-	RO	RO
0x134	CTITRIGOUTSTATUS	Debug	-	-	-	-	RO	RO
0x138	CTICHINSTAUS	Debug	-	-	-	-	RO	RO
0x13C	CTICHOUTSTATUS	Debug	-	-	-	-	RO	RO
0x140	CTIGATE	Debug	-	-	-	-	RW	RO
0xFC0	CTIDEVID2	Debug	-	-	-	-	RO	RO
0xFC4	CTIDEVID1	Debug	-	-	-	-	RO	RO
0xFC8	CTIDEVID	Debug	-	-	-	-	RO	RO

For the reset values of the CTI registers, see [Table E8-8](#).

E8.8 External debug register resets

Each register or field has a defined reset domain:

- Registers and fields in the Warm reset domain are also reset by a Cold reset and unchanged by an External Debug reset that is not coincident with a Cold reset or a Warm reset.
- Registers and fields in the Cold reset domain are unchanged by a Warm reset or an External Debug reset that is not coincident with a Cold reset.
- Registers and fields in the External Debug reset domain are unchanged by a Cold reset or a Warm reset that is not coincident with an External Debug reset.

A reset might change the value of a register. Specific rules apply to the observability of registers in the External Debug reset domain by indirect reads from the Core power domain when an External Debug reset is asserted without a coincident Cold reset. For more information, see [Synchronization of changes to the external debug registers](#).

[Table E8-7](#) and [Table E8-8](#) show the external debug register and CTI register resets. For other debug registers, see Management register resets.

Note

By reference to [Figure E6-1](#) the power domain can be deduced from the reset domain.

[Table E8-7](#) and [Table E8-8](#) do not include:

- Read-only identification registers, such as Processor ID Registers and [PMCFGR](#), that have a fixed value from reset.
- Read-only status registers, such as [EDSCR.RW](#), that are evaluated each time the register is read and that have no meaningful reset value.
- Write-only registers, such as [EDRCR](#), that only have an effect on writes, and have no meaningful reset value.
- Read/write registers, such as breakpoint and watchpoint registers, and [EDPRCR.CORENPDRQ](#), that alias other registers. The reset values are described by the descriptions of those other registers.
- IMPLEMENTATION DEFINED registers. The reset values and reset domains of these registers are also IMPLEMENTATION DEFINED and might be UNKNOWN.

All other fields in the registers are set to an IMPLEMENTATION DEFINED value that can be UNKNOWN. The register is in the specified reset domain.

Note

An IMPLEMENTATION DEFINED reset value, which can be UNKNOWN, means that hardware is not required to reset the register on the specified reset, but software must not rely on the register being preserved over reset.

Table E8-7 Summary of external debug register resets, debug registers

Register	Reset domain	Field	Value	Description
DBGPRCR	Cold	CORENPDRQ	The value of the powerup request ^a	Debug Power Control Register.
EDES	Warm	SS	EDECR.SS	Halting Step debug event pending
		RC	EDECR.RCE	Reset Catch debug event pending
		OSUC	0	OS Unlock Catch debug event pending
EDECR,	External debug	SS	0	Halting Step debug event enable
		RCE	0	Reset Catch debug event enable

Register	Reset domain	Field	Value	Description
		OSUCE	0	OS Unlock Catch debug event enable
EDWAR	Cold	-	-	All fields
EDSCR	Cold	RXfull	0	DTRRX register full
		TXfull	0	DTRTX register full
		RXO	0	DTRRX overrun
		TXU	0	DTRTX underrun
		INTdis	0	Interrupt disable
		TDA	0	Trap debug register accesses to Debug state
		MA	0	Memory access mode in Debug state
		HDE	0	Halting debug mode enable
		ERR	0	Cumulative error flag
EDECCR	Cold	NSE[2:1]	0b00	Coarse-grained Non-secure Exception Catch
		SE[3,1]	0b00	Coarse-grained Secure Exception Catch
EDPRCR,	External debug	COREPURQ ^a	-	Core powerup request
EDPRSR	Warm	SDR	-	Sticky debug restart
	Cold	SPMAD	0	Sticky EPMAD error
		SDAD	0	Sticky EDAD error
	Warm	SR	1	Sticky reset status
	Cold	SPD	1	Sticky powerdown status

a. On a Cold reset, [DBGPRCR](#).CORENPDRQ resets to the value of [EDPRCR](#).COREPURQ. If an External Debug reset and a Cold reset coincide, both [EDPRCR](#).COREPURQ and [DBGPRCR](#).CORENPDRQ are reset to 0.

[Table E8-5](#) shows the reset values for the CTI registers

Table E8-8 Summary of external debug register resets, CTI registers

Register	Reset domain	Field	Value	Description
CTICONTROL	External debug	GLBEN	0	CTI global enable
CTIAPPSET	External debug	-	-	All fields
CTIINEN<n>	External debug	-	-	All fields
CTIOUTEN<n>	External debug	-	-	All fields
CTIGATE	External debug	-	-	All fields
ASICCTL	IMPLEMENTATION DEFINED	-	IMPLEMENTATION DEFINED	All of register

Chapter E9

External Debug Register Descriptions

This chapter provides a description of the external debug registers. It contains the following sections:

- [About the external debug registers](#)
- [External debug registers](#)
- [External trace registers](#)
- [Cross-Trigger Interface registers](#)

E9.1 About the external debug registers

The following sections describe the registers that are accessible through the external debug interface:

- [External debug registers.](#)
- [External trace registers.](#)
- [Cross-Trigger Interface registers.](#)

E9.2 External debug registers

This section describes the debug registers that are accessible through the external debug interface and are used for external debug.

E9.2.1 DBGAUTHSTATUS_EL1, Debug Authentication Status Register

The DBGAUTHSTATUS_EL1 characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configuration

When FEAT_DoPD is implemented, DBGAUTHSTATUS_EL1 is in the Core power domain. Otherwise, DBGAUTHSTATUS_EL1 is in the Debug power domain

External register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to AArch32 System register [DBGAUTHSTATUS\[31:0\]](#).

Attributes

DBGAUTHSTATUS_EL1 is a 32-bit register.

Field descriptions



Bits [31:12]

Reserved, RES0.

HNID, bits [11:10]

Hyp non-invasive debug.

HNID	Meaning
0b00	Separate Hyp enable not implemented.
0b10	Implemented and disabled. ExternalHypNoninvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalHypNoninvasiveDebugEnabled() == TRUE.

Only permitted values for this field are 0b10 and 0b11. All other values are reserved.

HID, bits [9:8]

Hyp invasive debug.

HID	Meaning
0b00	Separate Hyp enable not implemented.
0b10	Implemented and disabled. ExternalHypInvasiveDebugEnabled() == FALSE.

0b11	Implemented and enabled. <code>ExternalHypInvasiveDebugEnabled() == TRUE</code> .
------	---

Only permitted values for this field are 0b10 and 0b11. All other values are reserved.

SNID, bits [7:6]

Secure non-invasive debug.

SNID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. <code>ExternalSecureNoninvasiveDebugEnabled() == FALSE</code> .
0b11	Implemented and enabled. <code>ExternalSecureNoninvasiveDebugEnabled() == TRUE</code> .

For Armv8-R AArch32, the only permitted value for this field is 0b00.

All other values are reserved.

SID, bits [5:4]

Secure invasive debug.

SID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. <code>ExternalSecureInvasiveDebugEnabled() == FALSE</code> .
0b11	Implemented and enabled. <code>ExternalSecureInvasiveDebugEnabled() == TRUE</code> .

For Armv8-R AArch32, the only permitted value for this field is 0b00.

All other values are reserved.

NSNID, bits [3:2]

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. <code>ExternalNoninvasiveDebugEnabled() == FALSE</code> .
0b11	Implemented and enabled. <code>ExternalNoninvasiveDebugEnabled() == TRUE</code> .

For Armv8-R AArch32, the only permitted values for this field are 0b10 and 0b11.

All other values are reserved.

NSID, bits [1:0]

Non-secure invasive debug.

NSID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. <code>ExternalInvasiveDebugEnabled()</code> == FALSE.
0b11	Implemented and enabled. <code>ExternalInvasiveDebugEnabled()</code> == TRUE.

For Armv8-R AArch32, the only permitted values for this field are 0b10 and 0b11.

All other values are reserved.

Accessing DBGAUTHSTATUS_EL1

DBGAUTHSTATUS_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFB8	DBGAUTHSTATUS_EL1

Accesses to this register are RO.

E9.2.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 63

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

Configuration

DBGBCR<n>_EL1 is in the Core power domain

If breakpoint n is not implemented, then accesses to this register are:

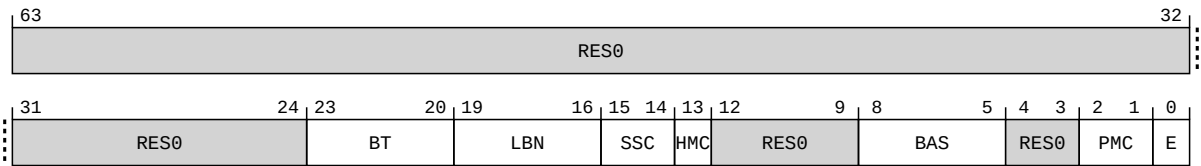
- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess().
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

External register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGBCR<n>[31:0].

Attributes

DBGBCR<n>_EL1 is a 64-bit register.

Field descriptions



When the E field is zero, all the other fields in the register are ignored.

Bits [63:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type.

Specifies breakpoint type.

BT	Meaning	Applies when
0b0000	Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.	
0b0001	Linked instruction address match. As 0b0000, but linked to a breakpoint that has linking enabled.	
0b0010	Unlinked Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR.	breakpoint n is context-aware
0b0011	As 0b0010, with linking enabled.	breakpoint n is context-aware

BT	Meaning	Applies when
0b0100	Unlinked instruction address mismatch. DBGBVR<n>_EL1 is the address of an instruction.	
0b0101	Linked instruction address mismatch. As 0b0100, but linked to a breakpoint that has linking enabled.	
0b1000	Unlinked VMID match. DBGBVR<n>_EL1 .VMID is a VMID compared against VSCTLR .VMID.	breakpoint n is context-aware
0b1001	As 0b1000, with linking enabled.	breakpoint n is context-aware
0b1010	Unlinked VMID and Context ID match. DBGBVR<n>_EL1 .ContextID is a Context ID compared against CONTEXTIDR_EL1 , and DBGBVR<n>_EL1 .VMID is a VMID compared against VSCTLR .VMID.	breakpoint n is context-aware
0b1011	As 0b1010, with linking enabled.	breakpoint n is context-aware

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked address matching breakpoints, specifies the index of the breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see 'Reserved [DBGBCR<n>_EL1](#).{SSC, HMC, PMC} values'.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see [DBGBCR<n>_EL1](#).SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

BAS, bits [8:5]

Byte address select. Defines which halfwords an address-matching breakpoint matches, regardless of the instruction set and Execution state.

The permitted values depend on the breakpoint type.

For Address match breakpoints in either AArch32 or AArch64 state, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0b0011	DBGBVR<n>_EL1	Use for T32 instructions.
0b1100	DBGBVR<n>_EL1 + 2	Use for T32 instructions.
0b1111	DBGBVR<n>_EL1	Use for A64 and A32 instructions.

All other values are reserved.

For more information, see ‘[Using the BAS field in Address Match breakpoints](#)’.

For Address mismatch breakpoints in an AArch32 stage 1 translation regime, the permitted values are:

BAS	Match instruction at	Constraint for debuggers
0b0000	•	Use for a match anywhere breakpoint.
0b0011	DBGBVR<n>_EL1	Use for stepping T32 instructions.
0b1100	DBGBVR<n>_EL1 + 2	Use for stepping T32 instructions.
0b1111	DBGBVR<n>_EL1	Use for stepping A64 and A32 instructions.

All other values are reserved.

For more information, see ‘[Using the BAS field in Address Match breakpoints](#)’.

For Context matching breakpoints, this field is RES1 and ignored.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the [DBGBCR<n>_EL1](#).SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see ‘Execution conditions for which a breakpoint generates Breakpoint exceptions’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable breakpoint n.

E	Meaning
0b0	Breakpoint n disabled.
0b1	Breakpoint n enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBCR<n>_EL1

This register is 32-bits wide and offset $0 \times 40C + (16 * n)$ is reserved.

DBGBCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	$0 \times 408 + (16 * n)$	DBGBCR<n>_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 63

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register [DBGBCR<n>_EL1](#).

Configuration

DBGBVR<n>_EL1 is in the Core power domain

How this register is interpreted depends on the value of [DBGBCR<n>_EL1.BT](#).

- When [DBGBCR<n>_EL1.BT](#) is 0b0x0x, this register holds a virtual address.
- When [DBGBCR<n>_EL1.BT](#) is 0b001x, 0b011x, or 0b110x, this register holds a Context ID.
- When [DBGBCR<n>_EL1.BT](#) is 0b100x, this register holds a VMID.
- When [DBGBCR<n>_EL1.BT](#) is 0b101x, this register holds a VMID and a Context ID.
- When [DBGBCR<n>_EL1.BT](#) is 0b111x, this register holds two Context ID values.

For other values of [DBGBCR<n>_EL1.BT](#), this register is RES0.

If breakpoint n is not implemented, then accesses to this register are:

- RES0 when `IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess()`.
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

External register DBGBVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register [DBGBCR<n>\[31:0\]](#).

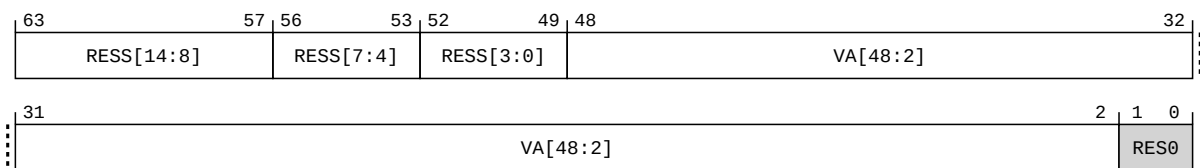
External register DBGBVR<n>_EL1 bits [63:32] are architecturally mapped to AArch32 System register [DBGXVR<n>\[31:0\]](#).

Attributes

DBGBVR<n>_EL1 is a 64-bit register.

Field descriptions

When [DBGBCR<n>_EL1.BT](#) IN {'0x0x'}:



RESS[14:8], bits [63:57]

Reserved, Sign extended. Software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

If the address is being matched in an AArch64 stage 1 translation regime:

- This field contains bits[48:2] of the address for comparison.

If the address is being matched in an AArch32 stage 1 translation regime, the first 20 bits of this field are RES0, and the rest of the field contains bits[31:2] of the address for comparison.

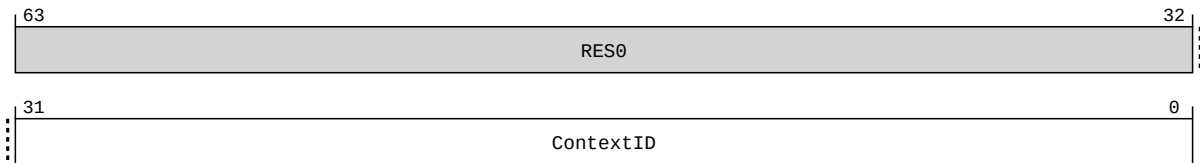
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When **DBGBCR<n>_EL1.BT** IN {'001x'}:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison.

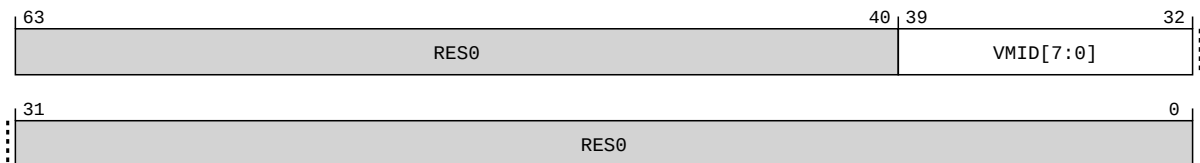
The value is compared against the following:

- [CONTEXTIDR](#) when the PE is executing at AArch32.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When **DBGBCR<n>_EL1.BT** IN {'100x'}:



Bits [63:40]

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

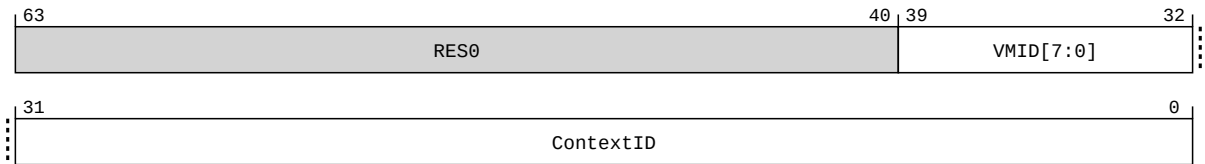
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'101x'}:



Bits [63:40]

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBVR<n>_EL1

DBGBVR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	0x400 + (16 * n)	DBGBVR<n>_EL1	63:0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear Register

The DBGCLAIMCLR_EL1 characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.
The architecture does not define any functionality for the CLAIM tag bits.

Note
CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the [DBGCLAIMSET_EL1](#) register.

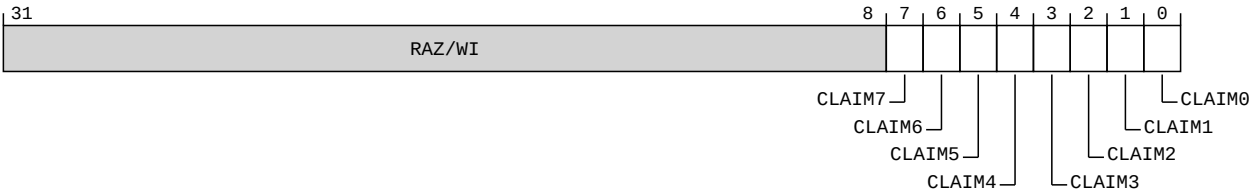
Configuration

DBGCLAIMCLR_EL1 is in the Core power domain
An implementation must include eight CLAIM tag bits.
External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register [DBGCLAIMCLR](#)[31:0].
External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register [DBGCLAIMSET](#)[31:0].
External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to External register [DBGCLAIMSET_EL1](#)[31:0].

Attributes

DBGCLAIMCLR_EL1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m>, bits [m], for m = 7 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.

CLAIM<m>	Meaning
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is W1C.

Accessing DBGCLAIMCLR_EL1

DBGCLAIMCLR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFA4	DBGCLAIMCLR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.5 **DBGCLAIMSET_EL1, Debug CLAIM Tag Set Register**

The DBGCLAIMSET_EL1 characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.
The architecture does not define any functionality for the CLAIM tag bits.

———— **Note** —————
CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the [DBGCLAIMCLR_EL1](#) register.

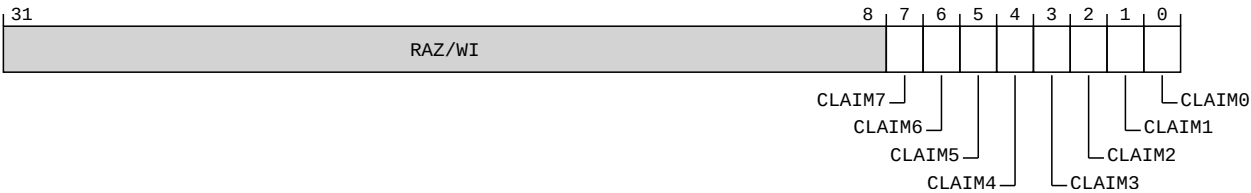
Configuration

DBGCLAIMSET_EL1 is in the Core power domain
An implementation must include eight CLAIM tag bits.
External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register [DBGCLAIMSET](#)[31:0].
External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register [DBGCLAIMCLR](#)[31:0].
External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to External register [DBGCLAIMCLR_EL1](#)[31:0].

Attributes

DBGCLAIMSET_EL1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Set. Used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a write: Ignored.
0b1	On a write: Set Claim Tag bit <m> to 1.

Access to this field is RAO/WIS.

Accessing DBGCLAIMSET_EL1

DBGCLAIMSET_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFA0	DBGCLAIMSET_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.6 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

Configuration

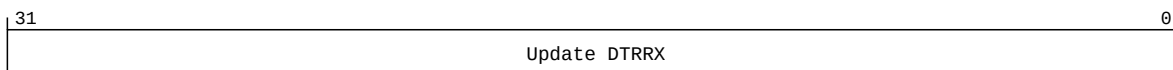
DBGDTRRX_EL0 is in the Core power domain

External register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to AArch32 System register [DBGDTRRXint\[31:0\]](#).

Attributes

DBGDTRRX_EL0 is a 32-bit register.

Field descriptions



DTRRX, bits [31:0]

Update DTRRX.

Writes to this register:

- If RXfull is 0, update the value in DTRRX and set RXfull to 1.
- If RXfull is 1, the written value is ignored and RXO is set to 1.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see ‘[The Debug Communication Channel and Instruction Transfer Register](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRX_EL0

If [EDSCR.ITE](#) == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in Memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

DBGDTRRX_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x080	DBGDTRRX_EL0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.7 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configuration

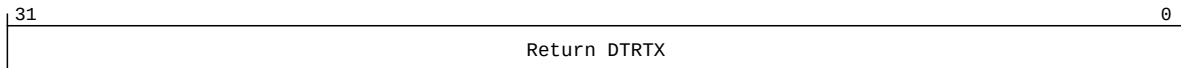
DBGDTRTX_EL0 is in the Core power domain

External register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to AArch32 System register [DBGDTRTXint](#)[31:0].

Attributes

DBGDTRTX_EL0 is a 32-bit register.

Field descriptions



DTRTX, bits [31:0]

Return DTRTX.

Reads of this register:

- If TXfull is 1, return the value in DTRTX and clear TXfull to 0.
- If TXfull is 0, return an UNKNOWN value and set TXU to 1.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see ‘[The Debug Communication Channel and Instruction Transfer Register](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTX_EL0

If [EDSCR](#).ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in Memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

DBGDTRTX_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x08C	DBGDTRTX_EL0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.8 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 63

The DBGWCR<n>_EL1 characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

Configuration

DBGWCR<n>_EL1 is in the Core power domain

If watchpoint n is not implemented, then accesses to this register are:

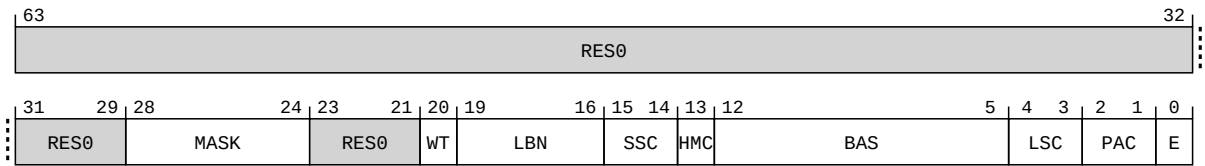
- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess(), RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

External register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register DBGWCR<n>[31:0].

Attributes

DBGWCR<n>_EL1 is a 64-bit register.

Field descriptions



Bits [63:29]

Reserved, RES0.

MASK, bits [28:24]

Address Mask. Only address ranges up to 2GB can be watched using a single mask.

MASK	Meaning
0b00000	No mask.
0b00011..0b11111	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from 0b00011 masking 3 address bits (0x00000007 mask for address) to 0b11111 masking 31 address bits (0x7FFFFFFF mask for address).

If programmed with a reserved value, the watchpoint behaves as if one of the following:

- DBGWCR<n>_EL1.MASK has been programmed with a defined value, which might be 0b00000 (no mask), other than for a direct read of DBGWCR<n>_EL1.
- The watchpoint is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [23:21]

Reserved, RES0.

WT, bit [20]

Watchpoint type. Possible values are:

WT	Meaning
0b0	Unlinked data address match.
0b1	Linked data address match.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked data address watchpoints, specifies the index of the breakpoint linked to.

For all other watchpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see ‘Execution conditions for which a watchpoint generates Watchpoint exceptions’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see ‘Execution conditions for which a watchpoint generates Watchpoint exceptions’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or doubleword addressed by [DBGWVR<n>_EL1](#) is being watched.

BAS	Description
xxxxxxx1	Match byte at DBGWVR<n>_EL1
xxxxxx1x	Match byte at DBGWVR<n>_EL1 + 1
xxxxx1xx	Match byte at DBGWVR<n>_EL1 + 2
xxxx1xxx	Match byte at DBGWVR<n>_EL1 + 3

In cases where [DBGWVR<n>_EL1](#) addresses a doubleword:

BAS	Description, if DBGWVR<n>_EL1[2] == 0
xxx1xxxx	Match byte at DBGWVR<n>_EL1 + 4
xx1xxxxx	Match byte at DBGWVR<n>_EL1 + 5
x1xxxxxx	Match byte at DBGWVR<n>_EL1 + 6
1xxxxxxx	Match byte at DBGWVR<n>_EL1 + 7

If [DBGWVR<n>_EL1\[2\] == 1](#), only BAS[3:0] is used. Arm deprecates setting [DBGWVR<n>_EL1\[2\] == 1](#).

The valid values for BAS are nonzero binary number all of whose set bits are contiguous. All other values are reserved and must not be used by software. See ‘[Reserved DBGWCR<n>.BAS values](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

LSC	Meaning
0b01	Match instructions that load from a watchpointed address.
0b10	Match instructions that store to a watchpointed address.
0b11	Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see ‘[Execution conditions for which a watchpoint generates Watchpoint exceptions](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable watchpoint n.

E	Meaning
0b0	Watchpoint n disabled.
0b1	Watchpoint n enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGWCR<n>_EL1

This register is 32-bits wide and offset $0 \times 80C + (16 * n)$ is reserved.

DBGWCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	$0 \times 808 + (16 * n)$	DBGWCR<n>_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.9 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 63

The DBGWVR<n>_EL1 characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register [DBGWCR<n>_EL1](#).

Configuration

DBGWVR<n>_EL1 is in the Core power domain

If watchpoint n is not implemented, then accesses to this register are:

- When `IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess(), RES0`.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

External register DBGWVR<n>_EL1 bits [31:0] are architecturally mapped to AArch32 System register [DBGWVR<n>\[31:0\]](#).

Attributes

DBGWVR<n>_EL1 is a 64-bit register.

Field descriptions

63				57		56		53		52		49		48												32								
RESS[14:8]										RESS[7:4]					RESS[3:0]					VA[48:2]														
31																															2		1	0
VA[48:2]																														RES0				

RESS[14:8], bits [63:57]

Reserved, Sign extended. Hardware and software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

Arm deprecates setting [DBGWVR<n>_EL1\[2\] = 1](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGWVR<n>_EL1

DBGWVR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	$0 \times 800 + (16 * n)$	DBGWVR<n>_EL1	63:0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.10 EDAA32PFR, External Debug Auxiliary Processor Feature Register

The EDAA32PFR characteristics are:

Purpose

Provides information about implemented PE features.

Note

The register mnemonic, EDAA32PFR, is derived from previous definitions of this register that defined this register only when AArch64 was not supported.

For general information about the interpretation of the ID registers, see ‘Principles of the ID scheme for fields in ID registers’.

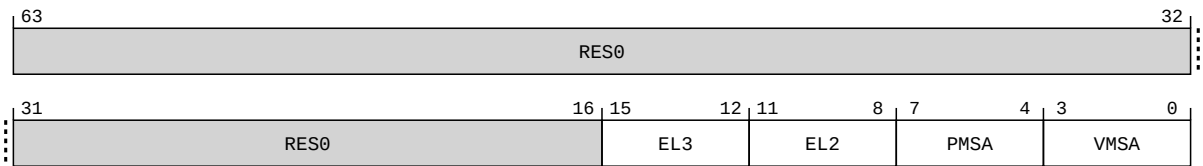
Configuration

There are no configuration notes.

Attributes

EDAA32PFR is a 64-bit register.

Field descriptions



Bits [63:16]

Reserved, RES0.

EL3, bits [15:12]

When EDPFR.EL3 == '0000':

AArch32 EL3 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL3	Meaning
0b0000	EL3 is not implemented or can be executed in AArch64 state.
0b0001	EL3 can be executed in AArch32 state only.

All other values are reserved.

In an Armv8-R AArch32 implementation this field returns the value of 0b0000.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

EL2, bits [11:8]

When **EDPFR.EL2** == '0000':

AArch32 EL2 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL2	Meaning
0b0000	EL2 is not implemented or can be executed in AArch64 state.
0b0001	EL2 can be executed in AArch32 state only.

All other values are reserved.

In an Armv8-R AArch32 implementation this field returns the value of 0b0001.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

PMSA, bits [7:4]

Indicates support for a 32-bit PMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMSA	Meaning
0b0000	PMSA-32 not supported.
0b0100	PMSAv8-32 supported.

In Armv8-R AArch32, the only permitted value is 0b0100. All other values are reserved.

Access to this field is RO.

VMSA, bits [3:0]

When **EDAA32PFR.PMSA** != '0000':

Indicates support for a VMSA in addition to a 32-bit PMSA.

VMSA	Meaning
0b0000	VMSA not supported.

All other values are reserved.

Access to this field is RO.

When **EDAA32PFR.PMSA** == '0000':

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMSA	Meaning
0b0000	VMSAv8-64 supported.
0b1111	Memory system architecture described by EDAA32PFR.MSA_frac.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing EDAA32PFR

EDAA32PFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD60	EDAA32PFR

Accessible as follows:

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

E9.2.11 EDACR, External Debug Auxiliary Control Register

The EDACR characteristics are:

Purpose

Allows implementations to support IMPLEMENTATION DEFINED controls.

Configuration

When FEAT_DoPD is implemented, EDACR is in the Core power domain. Otherwise, it is IMPLEMENTATION DEFINED whether EDACR is implemented in the Core power domain or in the Debug power domain

If this register is implemented, EDDEVID.AuxRegs == 0b0001.

If FEAT_DoPD is implemented, any mechanism to preserve control bits in EDACR over power down is optional and IMPLEMENTATION DEFINED.

If FEAT_DoPD is not implemented and EDACR contains any control bits that must be preserved over power down, then these bits must be accessible by the external debug interface when the OS Lock is locked, OSLSR_EL1.OSLK == 1, and when the Core is powered off.

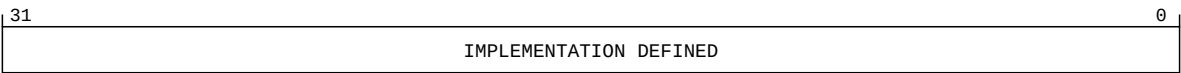
Changing this register from its reset value causes IMPLEMENTATION DEFINED behavior, including possible deviation from the architecturally-defined behavior.

Implementation of this register is OPTIONAL.

Attributes

EDACR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
 - On a Cold reset, this field resets to an architecturally UNKNOWN value.
 - On an External debug reset, the value of this field is unchanged.
 - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
 - On a Cold reset, the value of this field is unchanged.
 - On an External debug reset, this field resets to an architecturally UNKNOWN value.
 - On a Warm reset, the value of this field is unchanged.

Accessing EDACR

EDACR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x094	EDACR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.12 EDCIDR0, External Debug Component Identification Register 0

The EDCIDR0 characteristics are:

Purpose

Provides information to identify an external debug component.
For more information, see ‘About the Component Identification scheme’.

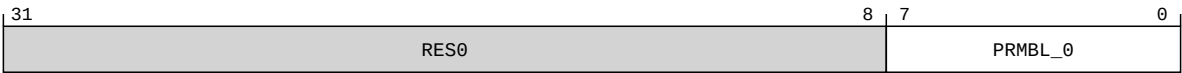
Configuration

When FEAT_DoPD is implemented, EDCIDR0 is in the Core power domain. Otherwise, EDCIDR0 is in the Debug power domain

Attributes

EDCIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

When FEAT_ED_PIDR is implemented:

Preamble.
Reads as 0x0D
Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing EDCIDR0

EDCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFF0	EDCIDR0

Accesses to this register are RO.

E9.2.13 EDCIDR1, External Debug Component Identification Register 1

The EDCIDR1 characteristics are:

Purpose

Provides information to identify an external debug component.
For more information, see ‘About the Component Identification scheme’.

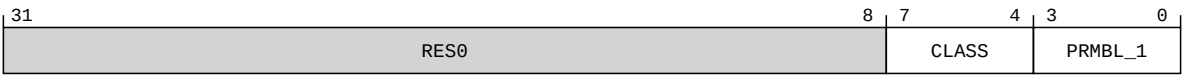
Configuration

When FEAT_DoPD is implemented, EDCIDR1 is in the Core power domain. Otherwise, EDCIDR1 is in the Debug power domain

Attributes

EDCIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

When FEAT_ED_PIDR is implemented:
Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.
This field reads as 0x9.
Access to this field is RO.

Otherwise:

Reserved, RAZ.

PRMBL_1, bits [3:0]

When FEAT_ED_PIDR is implemented:
Preamble.
Reads as 0b0000
Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing EDCIDR1

EDCIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFF4	EDCIDR1

Accesses to this register are RO.

E9.2.14 EDCIDR2, External Debug Component Identification Register 2

The EDCIDR2 characteristics are:

Purpose

Provides information to identify an external debug component.
For more information, see ‘About the Component Identification scheme’.

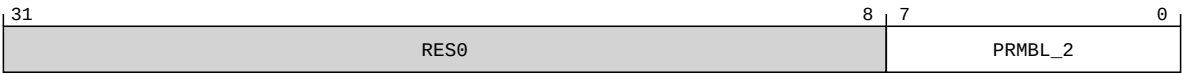
Configuration

When FEAT_DoPD is implemented, EDCIDR2 is in the Core power domain. Otherwise, EDCIDR2 is in the Debug power domain

Attributes

EDCIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

When FEAT_ED_PIDR is implemented:

Preamble.
Reads as 0x05
Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing EDCIDR2

EDCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFF8	EDCIDR2

Accesses to this register are RO.

E9.2.15 EDCIDR3, External Debug Component Identification Register 3

The EDCIDR3 characteristics are:

Purpose

Provides information to identify an external debug component.
For more information, see ‘About the Component Identification scheme’.

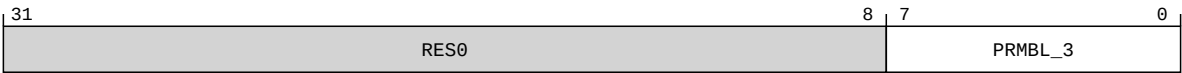
Configuration

When FEAT_DoPD is implemented, EDCIDR3 is in the Core power domain. Otherwise, EDCIDR3 is in the Debug power domain

Attributes

EDCIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

When FEAT_ED_PIDR is implemented:

Preamble.
Reads as 0xB1
Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing EDCIDR3

EDCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFFC	EDCIDR3

Accesses to this register are RO.

E9.2.16 EDCIDSR, External Debug Context ID Sample Register

The EDCIDSR characteristics are:

Purpose

Contains the sampled value of the Context ID, captured on reading [EDPCSR](#)[31:0].

Configuration

EDCIDSR is in the Core power domain

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

Attributes

EDCIDSR is a 32-bit register.

Field descriptions



CONTEXTIDR, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent [EDPCSR](#) sample. When the most recent [EDPCSR](#) sample is generated:

- If EL1 is using AArch32, then the Context ID is sampled from [CONTEXTIDR](#).

Because the value written to EDCIDSR is an indirect read of CONTEXTIDR, it is **CONSTRAINED UNPREDICTABLE** whether EDCIDSR is set to the original or new value if [EDPCSR](#) samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally **UNKNOWN** value.

Accessing EDCIDSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register **UNKNOWN**, see ‘[Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN](#)’.

EDCIDSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x0A4	EDCIDSR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an **ERROR**.
- Otherwise, accesses to this register are **RO**.

E9.2.17 EDDEVAFF0, External Debug Device Affinity register 0

The EDDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

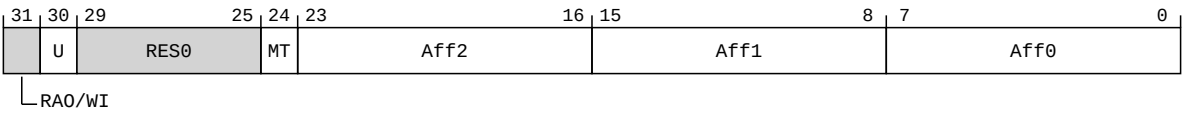
Configuration

When FEAT_DoPD is implemented, EDDEVAFF0 is in the Core power domain. Otherwise, EDDEVAFF0 is in the Debug power domain

Attributes

EDDEVAFF0 is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the [MPIDR](#).{Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDDEVAFF0

EDDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFA8	EDDEVAFF0

Accesses to this register are RO.

E9.2.18 EDDEVAFF1, External Debug Device Affinity register 1

The EDDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

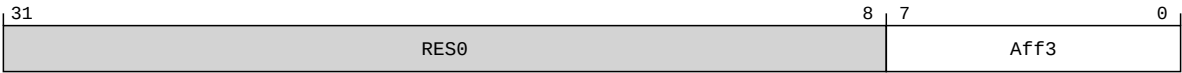
Configuration

When FEAT_DoPD is implemented, EDDEVAFF1 is in the Core power domain. Otherwise, EDDEVAFF1 is in the Debug power domain

Attributes

EDDEVAFF1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of [EDDEVAFF0.Aff0](#) for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDDEVAFF1

EDDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFAC	EDDEVAFF1

Accesses to this register are RO.

E9.2.19 EDDEVARCH, External Debug Device Architecture Register

The EDDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the external debug component.

Configuration

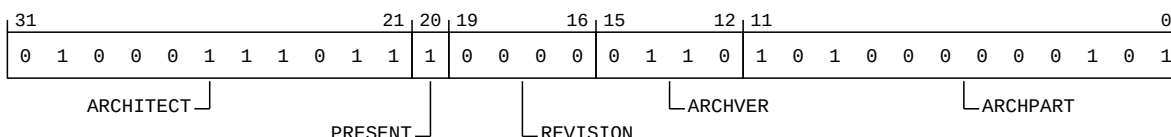
When FEAT_DoPD is implemented, EDDEVARCH is in the Core power domain. Otherwise, EDDEVARCH is in the Debug power domain

Attributes

EDDEVARCH is a 32-bit register.

Field descriptions

When FEAT_ED_PIDR is implemented:



ARCHITECT, bits [31:21]

Defines the architect of the component. For External Debug, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, 0b0100.

Bits [27:21] are the JEP106 identification code, 0b0111011.

Reads as 0b01000111011

Access to this field is RO.

PRESENT, bit [20]

DEVARCH presence. Indicates that the EDDEVARCH register provides architecture information.

Reads as 0b1

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For debug, the revision defined by Armv8 is 0x0.

All other values are reserved.

Reads as 0b0000

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EDDEVARCH.ARCHVER and EDDEVARCH.ARCHPART are also defined as a single field, EDDEVARCH.ARCHID, so that EDDEVARCH.ARCHVER is EDDEVARCH.ARCHID[15:12].

In Armv8-R AArch32, the only permitted value is 0b0110.

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

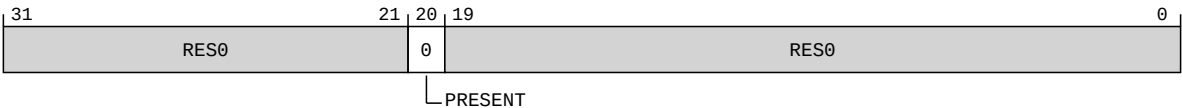
ARCHPART	Meaning
0xA05	Armv8-R AArch32 debug architecture.

EDDEVARCH.ARCHVER and EDDEVARCH.ARCHPART are also defined as a single field, EDDEVARCH.ARCHID, so that EDDEVARCH.ARCHPART is EDDEVARCH.ARCHID[11:0].

Armv8-R debug architecture.

Access to this field is RO.

Otherwise:



Bits [31:21]

Reserved, RES0.

PRESENT, bit [20]

DEVARCH presence. Indicates that the EDDEVARCH register does not provide architecture information.

Reads as 0b0

Access to this field is RO.

Bits [19:0]

Reserved, RES0.

Accessing EDDEVARCH

EDDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFBC	EDDEVARCH

Accesses to this register are RO.

E9.2.20 EDDEVID, External Debug Device ID register 0

The EDDEVID characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Configuration

When FEAT_DoPD is implemented, EDDEVID is in the Core power domain. Otherwise, EDDEVID is in the Debug power domain

Attributes

EDDEVID is a 32-bit register.

Field descriptions

31	28	27	24	23	4	3	0
RES0		AuxRegs		RES0		PCSample	

Bits [31:28]

Reserved, RES0.

AuxRegs, bits [27:24]

Indicates support for Auxiliary registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AuxRegs	Meaning
0b0000	None supported.
0b0001	Support for External Debug Auxiliary Control Register, EDACR .

All other values are reserved.

Access to this field is RO.

Bits [23:4]

Reserved, RES0.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSample	Meaning
0b0000	PC Sample-based Profiling Extension is not implemented in the external debug registers space.
0b0010	Only EDPCSR and EDCIDSR are implemented. This option is only permitted if EL3 and EL2 are not implemented.
0b0011	EDPCSR , EDCIDSR , and EDVIDSR are implemented.

All other values are reserved.

Access to this field is RO.

Accessing EDDEVID

EDDEVID can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC8	EDDEVID

Accesses to this register are RO.

E9.2.21 EDDEVID1, External Debug Device ID Register 1

The EDDEVID1 characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Configuration

When FEAT_DoPD is implemented, EDDEVID1 is in the Core power domain. Otherwise, EDDEVID1 is in the Debug power domain

Attributes

EDDEVID1 is a 32-bit register.

Field descriptions



Bits [31:4]

Reserved, RES0.

PCSR0ffset, bits [3:0]

Indicates the offset applied to PC samples returned by reads of [EDPCSR](#).

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSR0ffset	Meaning
0b0000	EDPCSR not implemented.
0b0010	EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state.

Access to this field is RO.

Accessing EDDEVID1

EDDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC4	EDDEVID1

Accesses to this register are RO.

E9.2.22 EDDEVID2, External Debug Device ID register 2

The EDDEVID2 characteristics are:

Purpose

Reserved for future descriptions of features of the debug implementation.

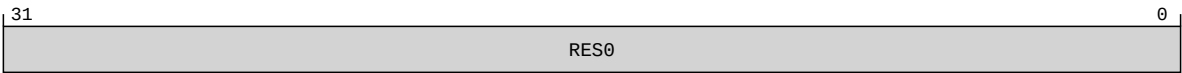
Configuration

When FEAT_DoPD is implemented, EDDEVID2 is in the Core power domain. Otherwise, EDDEVID2 is in the Debug power domain

Attributes

EDDEVID2 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing EDDEVID2

EDDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC0	EDDEVID2

Accesses to this register are RO.

E9.2.23 EDDEVTTYPE, External Debug Device Type register

The EDDEVTTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's debug logic.

Configuration

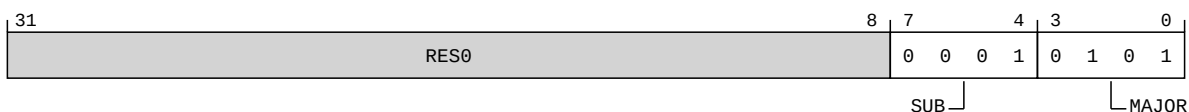
When FEAT_DoPD is implemented, EDDEVTTYPE is in the Core power domain. Otherwise, EDDEVTTYPE is in the Debug power domain

Attributes

EDDEVTTYPE is a 32-bit register.

Field descriptions

When FEAT_ED_PIDR is implemented:



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as 0b0001

Access to this field is RO.

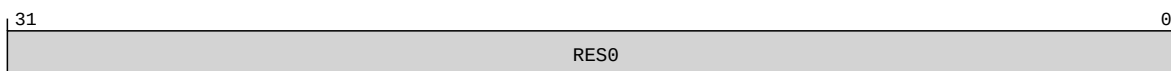
MAJOR, bits [3:0]

Major type. Indicates this is a debug logic component.

Reads as 0b0101

Access to this field is RO.

Otherwise:



Bits [31:0]

Reserved, RES0.

Accessing EDDEVTYPE

EDDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFCC	EDDEVTYPE

Accesses to this register are RO.

E9.2.24 EDDFR, External Debug Feature Register

The EDDFR characteristics are:

Purpose

Provides top-level information about the debug system.

Note

Debuggers must use [EDDEVARCH](#) to determine the Debug architecture version.

For general information about the interpretation of the ID registers, see ‘Principles of the ID scheme for fields in ID registers’.

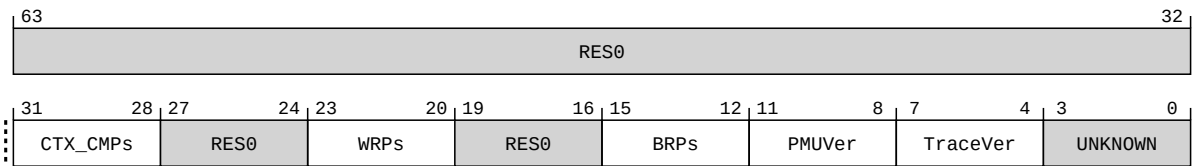
Configuration

There are no configuration notes.

Attributes

EDDFR is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

CTX_CMPS, bits [31:28]

Number of context-aware breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CTX_CMPS	Meaning
0b0000..0b1111	The number of context-aware breakpoints, minus 1.

The value of this field is never greater than EDDFR.BRPs.

Access to this field is RO.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WRPs	Meaning
0b0001..0b1111	The number of watchpoints, minus 1.

The value 0b0000 is reserved.

Access to this field is RO.

Bits [19:16]

Reserved, RES0.

BRPs, bits [15:12]

Number of breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BRPs	Meaning
0b0001..0b1111	The number of breakpoints, minus 1.

The value 0b0000 is reserved.

Access to this field is RO.

PMUVer, bits [11:8]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in ‘Alternative ID scheme used for the Performance Monitors Extension version’

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

[FEAT_PMUv3](#) implements the functionality identified by the value 0b0001.

Access to this field is RO.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a trace unit is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceVer	Meaning
0b0000	Trace unit System registers not implemented.

0b0001

Trace unit System registers implemented.

All other values are reserved.

A value of 0b0000 only indicates that no System register interface to a trace unit is implemented. A trace unit might nevertheless be implemented without a System register interface.

Access to this field is RO.

Bits [3:0]

Reserved, UNKNOWN.

Accessing EDDFR

EDDFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD28	EDDFR

Accessible as follows:

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

E9.2.25 EDECCR, External Debug Exception Catch Control Register

The EDECCR characteristics are:

Purpose

Controls Exception Catch debug events. For more information, see [‘Exception Catch debug event’](#).

Configuration

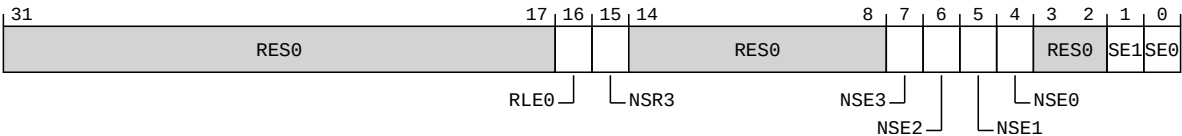
EDECCR is in the Core power domain

External register EDECCR bits [31:0] are architecturally mapped to AArch32 System register [DBGOSECCR](#)[31:0].

Attributes

EDECCR is a 32-bit register.

Field descriptions



Bits [31:17]

Reserved, RES0.

RLE0, bit [16]

Access to this field is RES0.

NSR3, bit [15]

Access to this field is RES0.

Bits [14:8]

Reserved, RES0.

NSE3, bit [7]

Access to this field is RES0.

NSE2, bit [6]

When Non-secure EL2 is implemented:

Coarse-grained exception catch for Non-secure EL2. Controls Exception Catch debug events for Non-secure EL2.

NSE2	Meaning
0b0	Exception Catch debug events are disabled for Non-secure EL2.

0b1	Exception Catch debug events are enabled for Non-secure EL2.
-----	--

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

NSE1, bit [5]

When Non-secure EL1 is implemented:

Coarse-grained exception catch for Non-secure EL1. Controls Exception Catch debug events for Non-secure EL1.

NSE1	Meaning
0b0	Exception Catch debug events are disabled for Non-secure EL1.
0b1	Exception Catch debug events are enabled for Non-secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

NSE0, bit [4]

Access to this field is RES0.

Bits [3:2]

Reserved, RES0.

SE1, bit [1]

When Secure EL1 is implemented:

Coarse-grained exception catch for Secure EL1. Controls Exception Catch debug events for Secure EL1.

SE1	Meaning
0b0	Exception Catch debug events are disabled for Secure EL1.
0b1	Exception Catch debug events are enabled for Secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

SE0, bit [0]

Access to this field is RES0.

Accessing EDECCR

EDECCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x098	EDECCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.26 EDECR, External Debug Execution Control Register

The EDECR characteristics are:

Purpose

Controls Halting debug events.

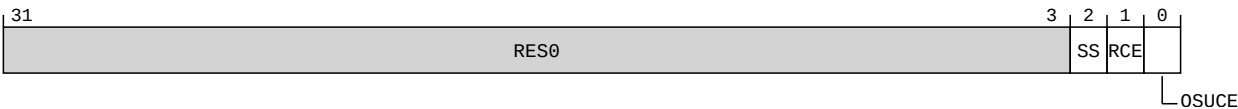
Configuration

When FEAT_DoPD is implemented, EDECR is in the Core power domain. Otherwise, EDECR is in the Debug power domain

Attributes

EDECR is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

SS, bit [2]

Halting step enable. Possible values of this field are:

SS	Meaning
0b0	Halting step debug event disabled.
0b1	Halting step debug event enabled.

If the value of EDECR.SS is changed when the PE is in Non-debug state, behavior is CONSTRAINED UNPREDICTABLE as described in [‘Changing the value of EDECR.SS when not in Debug state’](#).

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

RCE, bit [1]

Reset Catch Enable.

RCE	Meaning
0b0	Reset Catch debug event disabled.
0b1	Reset Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

OSUCE, bit [0]

OS Unlock Catch Enable.

OSUCE	Meaning
0b0	OS Unlock Catch debug event disabled.
0b1	OS Unlock Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Accessing EDECR

EDECR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x024	EDECR

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.27 EDESR, External Debug Event Status Register

The EDESR characteristics are:

Purpose

Indicates the status of internally pending Halting debug events.

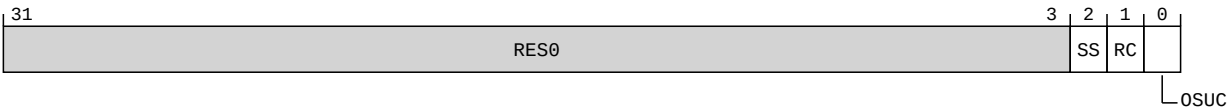
Configuration

EDESR is in the Core power domain

Attributes

EDESR is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

SS, bit [2]

Halting step debug event pending. Possible values of this field are:

SS	Meaning
0b0	Reading this means that a Halting step debug event is not pending. Writing this means no action.
0b1	Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in EDECR.SS.

RC, bit [1]

Reset Catch debug event pending. Possible values of this field are:

RC	Meaning
0b0	Reading this means that a Reset Catch debug event is not pending. Writing this means no action.
0b1	Reading this means that a Reset Catch debug event is pending. Writing this clears the pending Reset Catch debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in [EDECR.RCE](#).

OSUC, bit [0]

OS Unlock Catch debug event pending. Possible values of this field are:

OSUC	Meaning
0b0	Reading this means that an OS Unlock Catch debug event is not pending. Writing this means no action.
0b1	Reading this means that an OS Unlock Catch debug event is pending. Writing this clears the pending OS Unlock Catch debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing EDESR

If a request to clear a pending Halting debug event is received at or about the time when halting becomes allowed, it is **CONSTRAINED UNPREDICTABLE** whether the event is taken.

If Core power is removed while a Halting debug event is pending, it is lost. However, it might become pending again when the Core is powered back on and Cold reset.

EDESR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x020	EDESR

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.28 EDITCTRL, External Debug Integration mode Control register

The EDITCTRL characteristics are:

Purpose

Enables the external debug to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

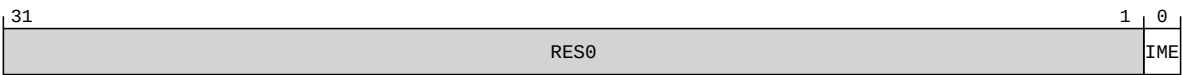
The power domain of EDITCTRL is IMPLEMENTATION DEFINED

Implementation of this register is OPTIONAL.

Attributes

EDITCTRL is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- Whichever power domain the register is implemented in, this field resets to 0.
- Otherwise, the value of this field is unchanged.

Accessing EDITCTRL

EDITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xF00	EDITCTRL

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.29 EDITR, External Debug Instruction Transfer Register

The EDITR characteristics are:

Purpose

Used in Debug state for passing instructions to the PE for execution.

Configuration

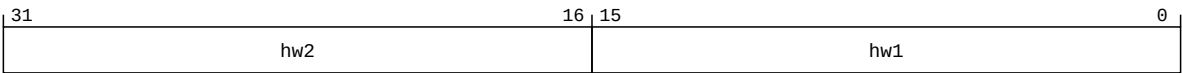
EDITR is in the Core power domain

Attributes

EDITR is a 32-bit register.

Field descriptions

When UsingAArch32():



hw2, bits [31:16]

Second halfword of the T32 instruction to be executed on the PE. When EDITR contains a 16-bit T32 instruction, this field is ignored. For more information, see ‘Behavior in Debug state’.

———— **Note** ————
The hw2 field is displayed on the left. This is not the usual convention for display of T32 instruction halfwords.

hw1, bits [15:0]

First halfword of the T32 instruction to be executed on the PE.

———— **Note** ————
The hw1 field is displayed on the right. This is not the usual convention for display of T32 instruction halfwords.

Accessing EDITR

IfEDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any instruction issued through the ITR in Normal access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

EDITR ignores writes if the PE is in Non-debug state.

EDITR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x084	EDITR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

E9.2.30 EDLAR, External Debug Lock Access Register

The EDLAR characteristics are:

Purpose

Allows or disallows access to the external debug registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

When FEAT_DoPD is implemented, EDLAR is in the Core power domain. Otherwise, EDLAR is in the Debug power domain

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE.

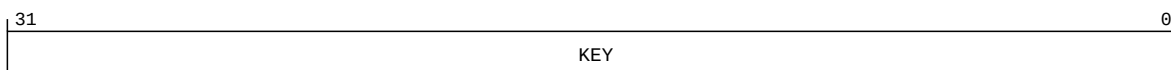
Software uses EDLAR to set or clear the lock, and [EDLSR](#) to check the current status of the lock.

Attributes

EDLAR is a 32-bit register.

Field descriptions

When Debug Software Lock is implemented:

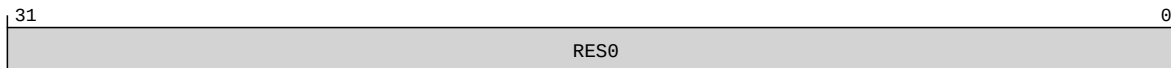


KEY, bits [31:0]

Lock Access control. Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:



Otherwise

Bits [31:0]

Reserved, RES0.

Accessing EDLAR

EDLAR can be accessed through the memory-mapped interface:

Component	Offset	Instance
Debug	0xFB0	EDLAR

Accesses to this register are WO.

E9.2.31 EDLSR, External Debug Lock Status Register

The EDLSR characteristics are:

Purpose

Indicates the current status of the software lock for external debug registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

When FEAT_DoPD is implemented, EDLSR is in the Core power domain. Otherwise, EDLSR is in the Debug power domain

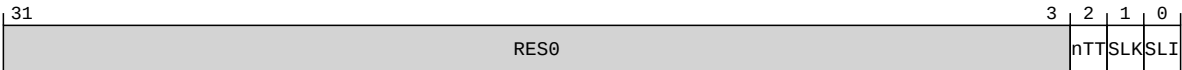
If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE.

Software uses [EDLAR](#) to set or clear the lock, and EDLSR to check the current status of the lock.

Attributes

EDLSR is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLK, bit [1]

When Debug Software Lock is implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when Software Lock is implemented, possible values of this field are:

SLK	Meaning
0b0	Lock clear. Writes are permitted to this component’s registers.
0b1	Lock set. Writes to this component’s registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a External debug reset, this field resets to '1'.

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
0b0	Software Lock not implemented or not memory-mapped access.
0b1	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing EDLSR

EDLSR can be accessed through the memory-mapped interface:

Component	Offset	Instance
Debug	0xFB4	EDLSR

Accesses to this register are RO.

E9.2.32 EDPCSR, External Debug Program Counter Sample Register

The EDPCSR characteristics are:

Purpose

Holds a sampled instruction address value.

Configuration

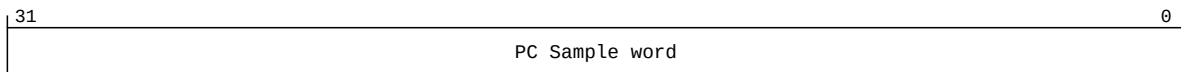
EDPCSR is in the Core power domain

If FEAT_Debugv8p1 is implemented, the format of this register differs depending on the value of [EDSCR.SC2](#).

Attributes

EDPCSR is a 32-bit register.

Field descriptions



EDPCSR, bits [31:0]

PC Sample. The sampled instruction address value.

EDPCSR reads as 0xFFFFFFFF when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If a branch instruction has retired since the PE left reset state, then the first read of EDPCSR is permitted but not required to return 0xFFFFFFFF.

EDPCSR reads as an UNKNOWN value when any of the following are true:

- The PE is in reset state.
- No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No branch instruction has retired since the last read of EDPCSR.

For the cases where a read of EDPCSR returns 0xFFFFFFFF or an UNKNOWN value, the read has the side-effect of setting [EDCIDSR](#), and [EDVIDSR](#) to UNKNOWN values.

Otherwise, a read of EDPCSR returns the sampled instruction address value and has the side-effect of indirectly writing to [EDCIDSR](#), and [EDVIDSR](#). The translation regime that EDPCSR samples can be determined from [EDVIDSR](#).{E2}.

For a read of EDPCSR from the memory-mapped interface, if [EDLSR.SLK](#) == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and [EDCIDSR](#), and [EDVIDSR](#) are unchanged.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing EDPCSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see ‘[Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN](#)’

In Armv8-R AArch32, EDPCSR is accessed as 32-bit memory mapped address.

EDPCSR can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	0x0A0	EDPCSR	31:0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Component	Offset	Instance	Range
Debug	0x0AC	EDPCSR	63:32

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

E9.2.33 EDPFR, External Debug Processor Feature Register

The EDPFR characteristics are:

Purpose

Provides information about implemented PE features.

For general information about the interpretation of the ID registers, see ‘Principles of the ID scheme for fields in ID registers’.

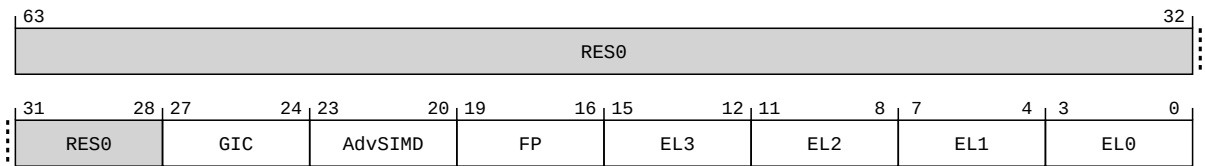
Configuration

There are no configuration notes.

Attributes

EDPFR is a 64-bit register.

Field descriptions



Bits [63:28]

Reserved, RES0.

GIC, bits [27:24]

System register GIC interface support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GIC	Meaning
0b0000	GIC CPU interface System registers not implemented.
0b0001	System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
0b0011	System register interface to version 4.1 of the GIC CPU interface is supported.

The table above shows all possible values of this field. The values this field can take in the current implementation are:

- 0b0000
- 0b0001

All other values are reserved.

In Armv8-R AArch32, the only permitted values are 0b0000 and 0b0001. All other values are reserved.

Access to this field is RO.

AdvSIMD, bits [23:20]

Advanced SIMD.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

This field must have the same value as the FP field.

[FEAT_AdvSIMD](#) implements the functionality identified by the value 0b0000.

In Armv8-R AArch32, the only permitted values are 0b0000 and 0b1111. All other values are reserved.

Access to this field is RO.

FP, bits [19:16]

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

This field must have the same value as the AdvSIMD field.

[FEAT_FP](#) implements the functionality identified by the value 0b0000.

In Armv8-R AArch32, the only permitted values are 0b0000 and 0b1111. All other values are reserved.

Access to this field is RO.

EL3, bits [15:12]

AArch64 EL3 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL3	Meaning
0b0000	EL3 is not implemented or cannot be executed in AArch64 state.
0b0001	EL3 can be executed in AArch64 state only.
0b0010	EL3 can be executed in both Execution states.

When the value of [EDAA32PFR.EL3](#) is nonzero, this field must be 0b0000.

All other values are reserved.

In an Armv8-R AArch32 implementation this field returns the value of 0b0000.

Access to this field is RO.

EL2, bits [11:8]

AArch64 EL2 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL2	Meaning
0b0000	EL2 is not implemented or cannot be executed in AArch64 state.
0b0001	EL2 can be executed in AArch64 state only.
0b0010	EL2 can be executed in both Execution states.

EL2	Meaning
-----	---------

When the value of [EDAA32PFR](#).EL2 is nonzero, this field must be 0b0000.

All other values are reserved.

In an Armv8-R AArch32 implementation this field returns the value of 0b0000.

Access to this field is RO.

EL1, bits [7:4]

AArch64 EL1 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL1	Meaning
0b0000	EL1 cannot be executed in AArch64 state. EL1 can be executed in AArch32 state only.
0b0001	EL1 can be executed in AArch64 state only.
0b0010	EL1 can be executed in both Execution states.

All other values are reserved.

In Armv8-R AArch32, the only permitted values is 0b0000.

Access to this field is RO.

EL0, bits [3:0]

AArch64 EL0 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL0	Meaning
0b0000	EL0 cannot be executed in AArch64 state. EL0 can be executed in AArch32 state only.
0b0001	EL0 can be executed in AArch64 state only.
0b0010	EL0 can be executed in both Execution states.

All other values are reserved.

In Armv8-R AArch32 implementation, the only permitted values is 0b0000.

Access to this field is RO.

Accessing EDPFR

EDPFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD20	EDPFR

Accessible as follows:

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

E9.2.34 **EDPIDR0, External Debug Peripheral Identification Register 0**

The EDPIDR0 characteristics are:

Purpose

Provides information to identify an external debug component.
For more information, see ‘About the Peripheral identification scheme’.

Configuration

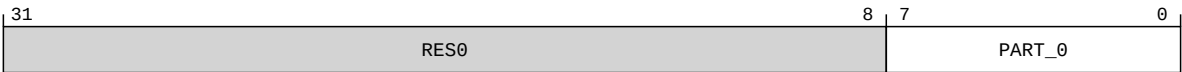
When FEAT_DoPD is implemented, EDPIDR0 is in the Core power domain. Otherwise, EDPIDR0 is in the Debug power domain

This register is present only when FEAT_ED_PIDR is implemented. Otherwise, direct accesses to EDPIDR0 are RES0.

Attributes

EDPIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
 - [EDPIDR1](#).PART_1 contains part number bits [11:8].
 - [EDPIDR0](#).PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
 - [EDPIDR1](#).PART_1 contains part number bits [15:12].
 - [EDPIDR0](#).PART_0 contains part number bits [11:4].
 - [EDPIDR2](#).REVISION contains part number bits [3:0].

When a 12-bit part number is used, [EDPIDR2](#).REVISION indicates revision information.
The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing EDPIDR0

EDPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFE0	EDPIDR0

Accesses to this register are RO.

E9.2.35 EDPIDR1, External Debug Peripheral Identification Register 1

The EDPIDR1 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR1 is in the Core power domain. Otherwise, EDPIDR1 is in the Debug power domain

This register is present only when FEAT_ED_PIDR is implemented. Otherwise, direct accesses to EDPIDR1 are RES0.

Attributes

EDPIDR1 is a 32-bit register.

Field descriptions

31		8	7	4	3	0	
RES0							
				DES_0		PART_1	

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- [EDPIDR1.DES_0](#): JEP106 identification code bits[3:0].
- [EDPIDR2.DES_1](#): JEP106 identification code bits[6:4].
- [EDPIDR4.DES_2](#): JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value $0 \times 7F$.
- A following 8-bit number, that is not $0 \times 7F$, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code $0 \times 7F \ 0 \times 7F \ 0 \times 7F \ 0 \times 7F \ 0 \times 3B$.

- The continuation code is the number of times $0 \times 7F$ appears before the final number. For example, a component designed by Arm Limited has the code 0×4 .
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code $0 \times 3B$.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
 - [EDPIDR1](#).PART_1 contains part number bits [11:8].
 - [EDPIDR0](#).PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
 - [EDPIDR1](#).PART_1 contains part number bits [15:12].
 - [EDPIDR0](#).PART_0 contains part number bits [11:4].
 - [EDPIDR2](#).REVISION contains part number bits [3:0].

When a 12-bit part number is used, [EDPIDR2](#).REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR1

EDPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFE4	EDPIDR1

Accesses to this register are RO.

E9.2.36 EDPIDR2, External Debug Peripheral Identification Register 2

The EDPIDR2 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

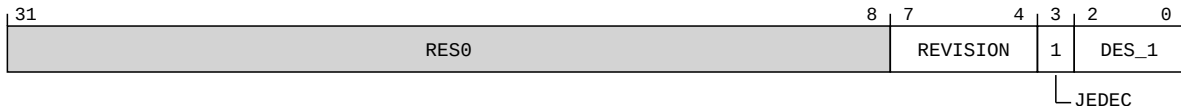
When FEAT_DoPD is implemented, EDPIDR2 is in the Core power domain. Otherwise, EDPIDR2 is in the Debug power domain

This register is present only when FEAT_ED_PIDR is implemented. Otherwise, direct accesses to EDPIDR2 are RES0.

Attributes

EDPIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- [EDPIDR2.REVISION](#) indicates the 4-bit revision number.
- [EDPIDR3.REVAND](#) indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- [EDPIDR2.REVISION](#) indicates the 4-bit major revision number.
- [EDPIDR3.REVAND](#) indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- [EDPIDR2.REVISION](#) contains part number bits [3:0].
- [EDPIDR3.REVAND](#) indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as 0b1

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- [EDPIDR1.DES_0](#): JEP106 identification code bits[3:0].
- [EDPIDR2.DES_1](#): JEP106 identification code bits[6:4].
- [EDPIDR4.DES_2](#): JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value 0x7F.
- A following 8-bit number, that is not 0x7F, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code 0x7F 0x7F 0x7F 0x7F 0x3B.

- The continuation code is the number of times 0x7F appears before the final number. For example, a component designed by Arm Limited has the code 0x4.
 - The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code 0x3B.
-

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR2

EDPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFE8	EDPIDR2

Accesses to this register are RO.

E9.2.37 EDPIDR3, External Debug Peripheral Identification Register 3

The EDPIDR3 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR3 is in the Core power domain. Otherwise, EDPIDR3 is in the Debug power domain

This register is present only when FEAT_ED_PIDR is implemented. Otherwise, direct accesses to EDPIDR3 are RES0.

Attributes

EDPIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- [EDPIDR2.REVISION](#) indicates the 4-bit revision number.
- [EDPIDR3.REVAND](#) indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- [EDPIDR2.REVISION](#) indicates the 4-bit major revision number.
- [EDPIDR3.REVAND](#) indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- [EDPIDR2.REVISION](#) contains part number bits [3:0].
- [EDPIDR3.REVAND](#) indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field

is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

[EDPIDR3](#).CMOD might also indicate component modifications.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of 0b0000 means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

[EDPIDR3](#).REVAND might also indicate component modifications.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR3

EDPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFEC	EDPIDR3

Accesses to this register are RO.

E9.2.38 EDPIDR4, External Debug Peripheral Identification Register 4

The EDPIDR4 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

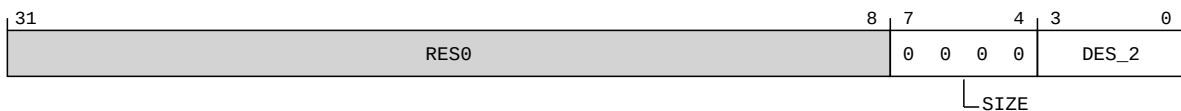
When FEAT_DoPD is implemented, EDPIDR4 is in the Core power domain. Otherwise, EDPIDR4 is in the Debug power domain

This register is present only when FEAT_ED_PIDR is implemented. Otherwise, direct accesses to EDPIDR4 are RES0.

Attributes

EDPIDR4 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. $\log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as 0b0000

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

- [EDPIDR1.DES_0](#): JEP106 identification code bits[3:0].
- [EDPIDR2.DES_1](#): JEP106 identification code bits[6:4].
- [EDPIDR4.DES_2](#): JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value 0x7F.
- A following 8-bit number, that is not 0x7F, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code 0x7F 0x7F 0x7F 0x7F 0x3B.

- The continuation code is the number of times 0x7F appears before the final number. For example, a component designed by Arm Limited has the code 0x4.
 - The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code 0x3B.
-

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR4

EDPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFD0	EDPIDR4

Accesses to this register are RO.

E9.2.39 **EDPIDR5, External Debug Peripheral Identification Register 5**

The EDPIDR5 characteristics are:

Purpose

Provides information to identify an external debug component.
For more information, see ‘About the Peripheral identification scheme’.

Configuration

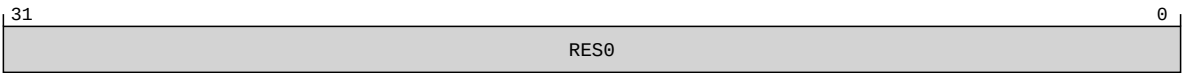
When FEAT_DoPD is implemented, EDPIDR5 is in the Core power domain. Otherwise, EDPIDR5 is in the Debug power domain

This register is present only when FEAT_ED_PIDR is implemented. Otherwise, direct accesses to EDPIDR5 are RES0.

Attributes

EDPIDR5 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing EDPIDR5

EDPIDR5 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFD4	EDPIDR5

Accesses to this register are RO.

E9.2.40 **EDPIDR6, External Debug Peripheral Identification Register 6**

The EDPIDR6 characteristics are:

Purpose

Provides information to identify an external debug component.
For more information, see ‘About the Peripheral identification scheme’.

Configuration

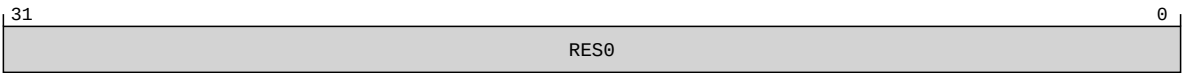
When FEAT_DoPD is implemented, EDPIDR6 is in the Core power domain. Otherwise, EDPIDR6 is in the Debug power domain

This register is present only when FEAT_ED_PIDR is implemented. Otherwise, direct accesses to EDPIDR6 are RES0.

Attributes

EDPIDR6 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing EDPIDR6

EDPIDR6 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFD8	EDPIDR6

Accesses to this register are RO.

E9.2.41 EDPIDR7, External Debug Peripheral Identification Register 7

The EDPIDR7 characteristics are:

Purpose

Provides information to identify an external debug component.
For more information, see ‘About the Peripheral identification scheme’.

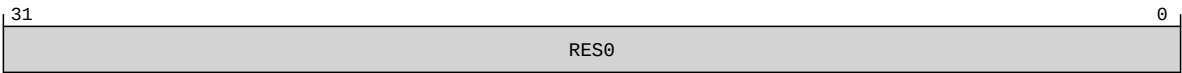
Configuration

When FEAT_DoPD is implemented, EDPIDR7 is in the Core power domain. Otherwise, EDPIDR7 is in the Debug power domain
This register is present only when FEAT_ED_PIDR is implemented. Otherwise, direct accesses to EDPIDR7 are RES0.

Attributes

EDPIDR7 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing EDPIDR7

EDPIDR7 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFDC	EDPIDR7

Accesses to this register are RO.

E9.2.42 EDPRCR, External Debug Power/Reset Control Register

The EDPRCR characteristics are:

Purpose

Controls the PE functionality related to powerup, reset, and powerdown.

Configuration

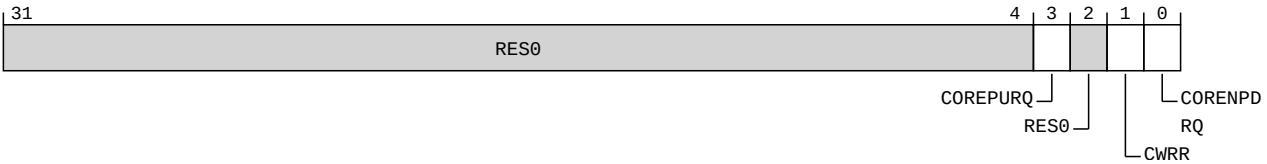
When FEAT_DoPD is implemented, EDPRCR is in the Core power domain. Otherwise, EDPRCR contains fields that are in the Core power domain and fields that are in the Debug power domain

External register EDPRCR bit [0] is architecturally mapped to AArch32 System register [DBGPRCR](#)[0].

Attributes

EDPRCR is a 32-bit register.

Field descriptions



Bits [31:4]

Reserved, RES0.

COREPURQ, bit [3]

Core powerup request. Allows a debugger to request that the power controller power up the core, enabling access to the debug register in the Core power domain, and that the power controller emulates powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

COREPURQ	Meaning
0b0	Do not request power up of the Core power domain.
0b1	Request power up of the Core power domain, and emulation of powerdown.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGPWRUPREQ** signal.

This field is in the Debug power domain and can be read and written when the Core power domain is powered off.

Note
Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RW.

Bit [2]

Reserved, RES0.

CWRR, bit [1]

Warm reset request.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

CWRR	Meaning
0b0	No action.
0b1	Request Warm reset.

This field is in the Core power domain

In an implementation that includes the recommended external debug interface, this bit drives the **DBGSTREQ** signal.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
 - !IsCorePowered()
 - DoubleLockStatus()
 - OSLockStatus()
 - SoftwareLockStatus(addrdesc)
- When !ExternalInvasiveDebugEnabled(), access to this field is RAZ/WI.
- When !ExternalHypInvasiveDebugEnabled(), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

This field is in the Core power domain, and permitted accesses to this field map to the [DBGPRCR.CORENPDRQ](#) and [DBGPRCR_EL1.CORENPDRQ](#) fields.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of [EDPRCR.COREPURQ](#) on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states, see ‘[Core power domain power states](#)’.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in [EDPRCR.COREPURQ](#).

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
 - !IsCorePowered()
 - DoubleLockStatus()
 - OSLockStatus()
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RW.

Accessing EDPRCR

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

EDPRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x310	EDPRCR

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.43 EDPRSR, External Debug Processor Status Register

The EDPRSR characteristics are:

Purpose

Holds information about the reset and powerdown state of the PE.

Configuration

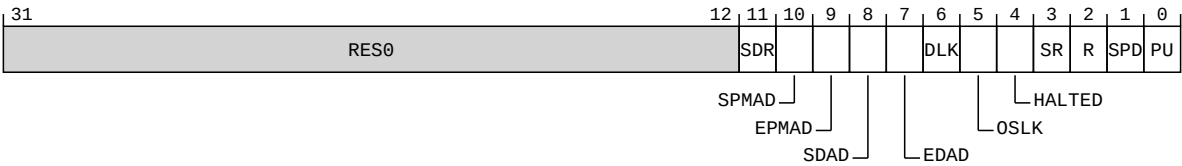
When FEAT_DoPD is implemented, EDPRSR is in the Core power domain. Otherwise, EDPRSR contains fields that are in the Core power domain and fields that are in the Debug power domain

If FEAT_DoPD is implemented, then all fields in this register are in the Core power domain.

Attributes

EDPRSR is a 32-bit register.

Field descriptions



Bits [31:12]

Reserved, RES0.

SDR, bit [11]

Sticky Debug Restart. Set to 1 when the PE exits Debug state.

SDR	Meaning
0b0	The PE has not restarted since EDPRSR was last read.
0b1	The PE has restarted since EDPRSR was last read.

Note

If a reset occurs when the PE is in Debug state, the PE exits Debug state. SDR is UNKNOWN on Warm reset, meaning a debugger must also use the SR bit to determine whether the PE has left Debug state.

If the Core power domain is powered up, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When `!IsCorePowered()`, access to this field is UNKNOWN/WI.
- When `DoubleLockStatus()`, access to this field is UNKNOWN/WI.
- When `SoftwareLockStatus(addrdesc)`, access to this field is RO.
- Otherwise, access to this field is RC/WI.

SPMAD, bit [10]

When FEAT_PMUv3_EXT is implemented:

Sticky EPMAD error.

SPMAD	Meaning
0b0	No external debug interface accesses to the Performance Monitors registers have failed because <code>AllowExternalPMUAccess() == FALSE</code> since EDPRSR was last read.
0b1	At least one external debug interface access to the Performance Monitors registers has failed and returned an error because <code>AllowExternalPMUAccess() == FALSE</code> since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If [FEAT_DoubleLock](#) is not implemented or `DoubleLockStatus() == FALSE`, this bit clears to 0.
- If [FEAT_DoubleLock](#) is implemented, and `DoubleLockStatus() == TRUE`, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When `!IsCorePowered()`, access to this field is UNKNOWN/WI.
- When `DoubleLockStatus()`, access to this field is UNKNOWN/WI.
- When `OSLockStatus()`, access to this field is UNKNOWN/WI.
- When `SoftwareLockStatus(addrdesc)`, access to this field is RO.
- Otherwise, access to this field is RC/WI.

Otherwise:

Reserved, RES0.

EPMAD, bit [9]

When FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Reports whether access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.EPMAD control.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMAD	Meaning
0b0	External access to Performance Monitor registers not affected. <code>AllowExternalPMUAccess() == TRUE</code> .

EPMAID	Meaning
0b1	External access to affected Performance Monitor registers is prohibited. <code>AllowExternalPMUAccess() == FALSE</code> .

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
 - `!IsCorePowered()`
 - `EDPRSR.R == '1'`
- When `DoubleLockStatus()`, access to this field is UNKNOWN/WI.
- When `OSLockStatus()`, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_PMUv3 is implemented:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

SDAD, bit [8]

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because `AllowExternalDebugAccess() == FALSE`.

SDAD	Meaning
0b0	No external debug interface accesses to the debug registers have failed because <code>AllowExternalDebugAccess() == FALSE</code> since EDPRSR was last read.
0b1	At least one external debug interface access to the debug registers has failed and returned an error because <code>AllowExternalDebugAccess() == FALSE</code> since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If [FEAT_DoubleLock](#) is not implemented or `DoubleLockStatus() == FALSE`, this bit clears to 0.
- If [FEAT_DoubleLock](#) is implemented and `DoubleLockStatus() == TRUE`, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
 - `!IsCorePowered()`
 - `EDPRSR.R == '1'`
- When `DoubleLockStatus()`, access to this field is UNKNOWN/WI.
- Access to this field is UNKNOWN/WI if all of the following are true:
 - `OSLockStatus()`
 - external debug writes to OSLAR_EL1 do not return an error when `AllowExternalDebugAccess(addrdesc) == FALSE`
- Otherwise, access to this field is RO.

EDAD, bit [7]

External Debug Access Disable status. Reports whether access to breakpoint registers, watchpoint registers, and optionally [OSLAR_EL1](#) by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
0b0	External access to debug registers not affected. <code>AllowExternalDebugAccess()</code> == TRUE.
0b1	External access to affected debug registers is prohibited. <code>AllowExternalDebugAccess()</code> == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
 - `!IsCorePowered()`
 - `EDPRSR.R == '1'`
- When `DoubleLockStatus()`, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

DLK, bit [6]

When FEAT_DoubleLock is implemented:

Double Lock.

This field returns the result of the pseudocode function `DoubleLockStatus()`.

If the Core power domain is powered up and `DoubleLockStatus() == TRUE`, it is IMPLEMENTATION DEFINED whether:

- `EDPRSR.PU` reads as 1, `EDPRSR.DLK` reads as 1, and `EDPRSR.SPD` is UNKNOWN.
- `EDPRSR.PU` reads as 0, `EDPRSR.DLK` is UNKNOWN, and `EDPRSR.SPD` reads as 0.

This field is in the Core power domain.

DLK	Meaning
0b0	<code>DoubleLockStatus()</code> returns FALSE.
0b1	<code>DoubleLockStatus()</code> returns TRUE and the Core power domain is powered up.

Accessing this field has the following behavior:

- When `!IsCorePowered()`, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

OSLK, bit [5]

OS Lock status bit.

A read of this bit returns the value of `OSLSR_EL1.OSLK`.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
 - !IsCorePowered()
 - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

HALTED, bit [4]

Halted status bit.

HALTED	Meaning
0b0	PE is in Non-debug state.
0b1	PE is in Debug state.

This field is in the Core power domain.

Accessing this field has the following behavior:

- When !IsCorePowered(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

SR, bit [3]

Sticky core Reset status bit.

SR	Meaning
0b0	The non-debug logic of the PE is not in reset state and has not been reset since the last time EDPRSR was read.
0b1	The non-debug logic of the PE is in reset state or has been reset since the last time EDPRSR was read.

If EDPRSR.PU reads as 1 and EDPRSR.R reads as 0, which means that the Core power domain is in a powerup state and that the non-debug logic of the PE is not in reset state, then following a read of EDPRSR:

- If [FEAT_DoubleLock](#) is not implemented or DoubleLockStatus() == FALSE, this bit clears to 0.
- If [FEAT_DoubleLock](#) is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Accessing this field has the following behavior:

- When !IsCorePowered(), access to this field is UNKNOWN/WI.
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is RC/WI.

R, bit [2]

PE Reset status bit.

R	Meaning
0b0	The non-debug logic of the PE is not in reset state.
0b1	The non-debug logic of the PE is in reset state.

If [FEAT_DoubleLock](#) is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information, see ‘[EDPRSR.{DLK, R} and reset state](#)’.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
 - !IsCorePowered()
 - DoubleLockStatus()
- Otherwise, access to this field is RO.

SPD, bit [1]

Sticky core Powerdown status bit.

If [FEAT_DoubleLock](#) is implemented and `DoubleLockStatus() == TRUE`, then:

- If [FEAT_Debugv8p2](#) is implemented, this bit reads as 0.
- If [FEAT_Debugv8p2](#) is not implemented, this bit might read as 0 or 1.

For more information, see ‘[EDPRSR.{DLK, SPD, PU} and the Core power domain](#)’.

SPD	Meaning
0b0	If <code>EDPRSR.PU</code> is 0, it is not known whether the state of the debug registers in the Core power domain is lost. If <code>EDPRSR.PU</code> is 1, the state of the debug registers in the Core power domain has not been lost.
0b1	The state of the debug registers in the Core power domain has been lost.

If the Core power domain is powered up, then, following a read of `EDPRSR`:

- If [FEAT_DoubleLock](#) is not implemented or `DoubleLockStatus() == FALSE`, this bit clears to 0.
- If [FEAT_DoubleLock](#) is implemented and `DoubleLockStatus() == TRUE`, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

`EDPRSR.{DLK, SPD, PU}` describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see ‘[EDPRSR.{DLK, SPD, PU} and the Core power domain](#)’.

When [FEAT_DoPD](#) is not implemented and the Core power domain is in either retention or powerdown state, the value of `EDPRSR.SPD` is IMPLEMENTATION DEFINED. For more information, see ‘[EDPRSR.SPD when the Core domain is in either retention or powerdown state](#)’.

The reset behavior of this field is:

- On a Cold reset, this field resets to '1'.

Accessing this field has the following behavior:

- When `!IsCorePowered()`, access to this field is RAZ/WI.

- Access to this field is UNKNOWN/WI if all of the following are true:
 - IsCorePowered()
 - DoubleLockStatus()
- Otherwise, access to this field is RC/WI.

PU, bit [0]

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

PU	Meaning
0b0	Core power domain is in a low-power or powerdown state where the debug registers in the Core power domain cannot be accessed.
0b1	Core power domain is in a powerup state where the debug registers in the Core power domain can be accessed.

If [FEAT_DoubleLock](#) is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information see ‘[EDPRSR.{DLK, R} and reset state](#)’

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see ‘[EDPRSR.{DLK, SPD, PU} and the Core power domain](#)’.

When the Core power domain is powered-up and `DoubleLockStatus() == TRUE`, then the value of EDPRSR.PU is IMPLEMENTATION DEFINED. See the description of the DLK bit for more information.

If [FEAT_DoubleLock](#) is implemented, the Core power domain is powered up, and `DoubleLockStatus() == TRUE`, it is IMPLEMENTATION DEFINED whether this bit reads as 0 or 1.

Access to this field is RO.

Accessing EDPRSR

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

If the Core power domain is powered up (EDPRSR.PU == 1), then following a read of EDPRSR:

- If [FEAT_DoubleLock](#) is not implemented or `DoubleLockStatus() == FALSE`, then:
 - EDPRSR.{SDR, SPMAD, SDAD, SPD} are cleared to 0.
 - EDPRSR.SR is cleared to 0 if the non-debug logic of the PE is not in reset state (EDPRSR.R == 0).
- If [FEAT_DoubleLock](#) is implemented and `DoubleLockStatus() == TRUE`, it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

If [FEAT_DoPD](#) is not implemented and the Core power domain is powered down (EDPRSR.PU == 0), then:

- EDPRSR.{SDR, SPMAD, SDAD, SR} are all UNKNOWN, and are either reset or restored on being powered up.
- EDPRSR.SPD is not cleared following a read of EDPRSR. See the SPD bit description for more information.

The clearing of bits is an indirect write to EDPRSR.

EDPRSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x314	EDPRSR

Accesses to this register are RO.

E9.2.44 EDRCR, External Debug Reserve Control Register

The EDRCR characteristics are:

Purpose

This register is used to allow imprecise entry to Debug state and clear sticky bits in [EDSCR](#).

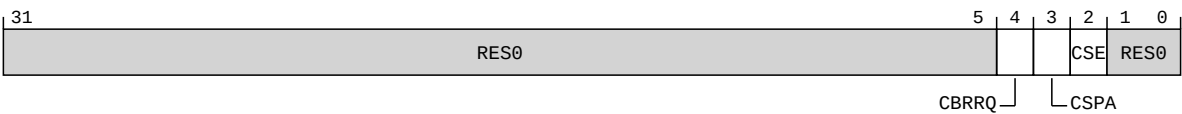
Configuration

EDRCR is in the Core power domain

Attributes

EDRCR is a 32-bit register.

Field descriptions



Bits [31:5]

Reserved, RES0.

CBRRQ, bit [4]

Allow imprecise entry to Debug state. The actions on writing to this bit are:

CBRRQ	Meaning
0b0	No action.
0b1	Allow imprecise entry to Debug state, for example by canceling pending bus accesses.

Setting this bit to 1 allows a debugger to request imprecise entry to Debug state. An External Debug Request debug event must be pending before the debugger sets this bit to 1.

This feature is optional. If this feature is not implemented, writes to this bit are ignored.

When !ExternalInvasiveDebugEnabled() or !ExternalHypInvasiveDebugEnabled(), access to this field is RAZ/WI.

CSPA, bit [3]

Clear Sticky Pipeline Advance. This bit is used to clear the [EDSCR](#).PipeAdv bit to 0. The actions on writing to this bit are:

CSPA	Meaning
0b0	No action.

CSPA	Meaning
0b1	Clear the EDSCR .PipeAdv bit to 0.

CSE, bit [2]

Clear Sticky Error. Used to clear the [EDSCR](#) cumulative error bits to 0. The actions on writing to this bit are:

CSE	Meaning
0b0	No action.
0b1	Clear the EDSCR .{TXU, RXO, ERR} bits, and, if the PE is in Debug state, the EDSCR .ITO bit, to 0.

Bits [1:0]

Reserved, RES0.

Accessing EDRCR

EDRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x090	EDRCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

E9.2.45 EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

Purpose

Main control register for the debug implementation.

Configuration

EDSCR is in the Core power domain

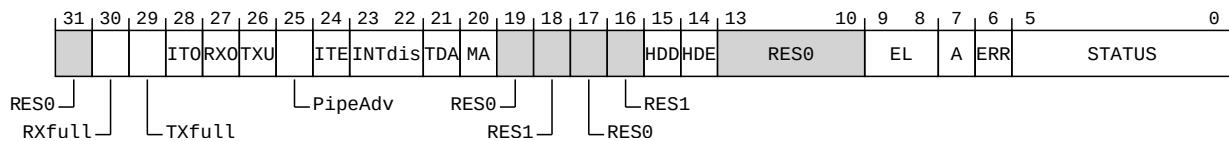
External register EDSCR bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to AArch32 System register [DBGDSCRext](#)[31:29, 27:26, 23:21, 19, 14, 6].

External register EDSCR bits [30:29] are architecturally mapped to AArch32 System register [DBGDSCRint](#)[30:29].

Attributes

EDSCR is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full.

When [DBGOSLSR.OSLK](#) is 1, this bit can be indirectly read and written through [DBGDSCRext](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

TXfull, bit [29]

DTRTX full.

When [DBGOSLSR.OSLK](#) is 1, this bit can be indirectly read and written through [DBGDSCRext](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

ITO, bit [28]

ITR overrun. Set to 0 on entry to Debug state.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.

- Otherwise, access to this field is RO.

RXO, bit [27]

DTRRX overrun.

When [DBGOSLSR.OSLK](#) is 1, this bit can be indirectly read and written through [DBGDSCRext](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

TXU, bit [26]

DTRTX underrun.

When [DBGOSLSR.OSLK](#) is 1, this bit can be indirectly read and written through [DBGDSCRext](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

PipeAdv, bit [25]

Pipeline Advance. Indicates that software execution is progressing.

PipeAdv	Meaning
0b0	No progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA .
0b1	Progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA .

The architecture does not define precisely when this field is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing. For example, a PE might set this field to 1 each time the PE retires one or more instructions, or at periodic intervals during the progression of an instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

ITE, bit [24]

ITR empty.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

INTdis, bits [23:22]

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
0b00	Masking of interrupts and SError exceptions is controlled by PSTATE and interrupt routing controls.
0b01	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure EL1 are masked.
0b10	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure EL1 are masked.
0b11	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked.

Note

This control affects both physical and virtual interrupts and SError exceptions.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

- [DBGDSCRext](#).

The Effective value of this field is 0b00 when ExternalInvasiveDebugEnabled() is FALSE.

Support for the values 0b01 and 0b10 is IMPLEMENTATION DEFINED. If these values are not supported, they are reserved. If programmed with a reserved value, the PE behaves as if EDSCR.INTdis has been programmed with a defined value, other than for a direct read of EDSCR, and the value returned by a read of EDSCR.INTdis is UNKNOWN.

The reset behavior of this field is:

- On a Cold reset, this field resets to '00'.

TDA, bit [21]

Traps accesses to the following debug registers:

- AArch32: [DBGBCR<n>](#), [DBGBVR<n>](#), [DBGBXVR<n>](#), [DBGWCR<n>](#), [DBGWVR<n>](#).

TDA	Meaning
0b0	Accesses to debug registers do not generate a Software Access Debug event.
0b1	Accesses to debug registers generate a Software Access Debug event, if OSLSR_EL1.OSLK is 0 and if halting is allowed.

When [DBGOSLSR](#).OSLK is 1, this bit can be indirectly read and written through [DBGDSCRext](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

MA, bit [20]

Memory access mode. Controls the use of Memory access mode for accessing ITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:

MA	Meaning
0b0	Normal access mode.
0b1	Memory access mode.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Bit [19]

Reserved, RES0.

Bit [18]

Reserved, RES1.

Bit [17]

Reserved, RES0.

Bit [16]

Reserved, RES1.

HDD, bit [15]

Hyp debug disabled.

On entry to Debug state:

- If entering Debug state in Hyp mode, this bit is set to 0.
- Otherwise, this bit is set to the inverse of `ExternalHypInvasiveDebugEnabled()`.

In Debug state, the value of the HDD bit does not change, even if `ExternalHypInvasiveDebugEnabled()` changes.

In Non-debug state, HDD returns the inverse of `ExternalHypInvasiveDebugEnabled()`.

If the authentication signals that control `ExternalHypInvasiveDebugEnabled()` change, a context synchronization event is required to guarantee their effect.

Access to this field is RO.

HDE, bit [14]

Halting debug enable.

HDE	Meaning
0b0	Halting disabled for Breakpoint, Watchpoint and Halt Instruction debug events.
0b1	Halting enabled for Breakpoint, Watchpoint and Halt Instruction debug events.

When [DBGOSLSR.OSLK](#) is 1, this bit can be indirectly read and written through [DBGDSCRext](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Bits [13:10]

Reserved, RES0.

EL, bits [9:8]

Exception level. In Debug state, gives the current Exception level of the PE.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

A, bit [7]

SError exception pending. In Debug state, indicates whether an SError exception is pending:

- If PE is executing at EL0 or EL1, and [HCR](#).{AMO, TGE} is {1, 0}, a virtual SError exception.
- Otherwise, a physical SError exception.

A	Meaning
0b0	No SError exception pending.
0b1	SError exception pending.

A debugger can read EDSCR to check whether an SError exception is pending without having to execute further instructions. A pending SError might indicate data from target memory is corrupted.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

ERR, bit [6]

Cumulative error flag. This bit is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

When [DBGOSLSR](#).OSLK is 1, this bit can be indirectly read and written through [DBGDSCRExt](#).

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

STATUS, bits [5:0]

On entering Debug state, the PE sets this field to indicate the reason for halting.

STATUS	Meaning
0b000001	PE is restarting, exiting Debug state.
0b000010	PE is in Non-debug state.
0b000111	Breakpoint.
0b010011	External debug request.

STATUS	Meaning
0b011011	Halting step, normal.
0b011111	Halting step, exclusive.
0b100011	OS Unlock Catch.
0b100111	Reset Catch.
0b101011	Watchpoint.
0b101111	HLT instruction.
0b110011	Software access to debug register.
0b110111	Exception Catch.
0b111011	Halting step, no syndrome.

All other values are reserved.

The PE resets into Non-debug state. However, if the PE enters Debug state immediately after reset, then the reset value is overwritten with the reason for halting.

Arm deprecates software use of the Halting step, normal and Halting step, exclusive values.

The reset behavior of this field is:

- On a Cold reset, this field resets to '000010'.

Access to this field is RO.

Accessing EDSCR

EDSCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x088	EDSCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.2.46 EDVIDSR, External Debug Virtual Context Sample Register

The EDVIDSR characteristics are:

Purpose

Contains sampled values captured on reading [EDPCSR](#)[31:0].

Configuration

EDVIDSR is in the Core power domain

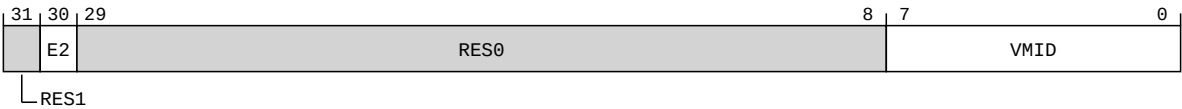
Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

When [FEAT_PCSRv8](#) is implemented in the external debug registers space, if EL2 is not implemented, it is IMPLEMENTATION DEFINED whether EDVIDSR is implemented.

Attributes

EDVIDSR is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES1.

E2, bit [30]

Exception level 2 status sample. Indicates whether the most recent [EDPCSR](#) sample was associated with EL2.

E2	Meaning
0b0	Sample is not from EL2.
0b1	Sample is from EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [29:8]

Reserved, RES0.

VMID, bits [7:0]

VMID sample. The VMID associated with the most recent [EDPCSR](#) sample. When the most recent [EDPCSR](#) sample was generated, this field is RES0 if the PE is executing at EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing EDVIDSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see [‘Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN’](#).

EDVIDSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x0A8	EDVIDSR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

E9.2.47 EDWAR, External Debug Watchpoint Address Register

The EDWAR characteristics are:

Purpose

Returns the virtual data address being accessed when a Watchpoint Debug Event was triggered.

Configuration

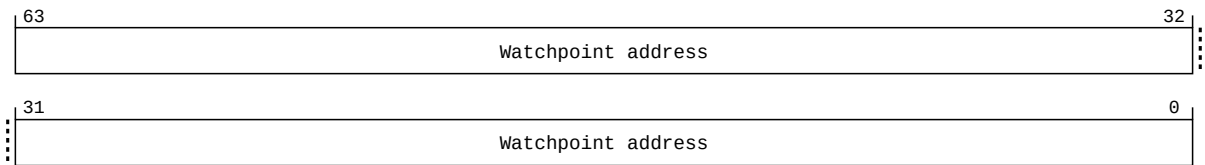
EDWAR is in the Core power domain

The value of this register is UNKNOWN if the PE is in Non-debug state, or if `EDSCR.STATUS` is not `0b101011`.

Attributes

EDWAR is a 64-bit register.

Field descriptions



ADDR, bits [63:0]

Watchpoint address. The data virtual address being accessed when a Watchpoint Debug Event was triggered and caused entry to Debug state. This address must be within a naturally-aligned block of memory of power-of-two size no larger than the DC ZVA block size.

The value of this register is UNKNOWN if the PE is in Non-debug state, or if Debug state was entered other than for a Watchpoint debug event.

The value of `EDWAR[63:32]` is UNKNOWN if Debug state was entered for a Watchpoint debug event taken from AArch32 state.

The EDWAR is subject to the same alignment rules as the reporting of a watchpointed address in the FAR. See ‘Exception syndrome information and preferred return address’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing EDWAR

EDWAR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x030	EDWAR

Accessible as follows:

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `OSLockStatus()`, accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

E9.2.48 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configuration

The power domain of MIDR_EL1 is IMPLEMENTATION DEFINED

External register MIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register [MIDR](#)[31:0].

Attributes

MIDR_EL1 is a 32-bit register.

Field descriptions

31	24	23	20	19	16	15	4	3	0
Implementer		Variant		Architecture		PartNum		Revision	

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Access to this field is RO.

Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MIDR_EL1

MIDR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD00	MIDR_EL1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register are IMPLEMENTATION DEFINED.
- Otherwise, accesses to this register are RO.

E9.2.49 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

Purpose

Used to lock or unlock the OS Lock.

Configuration

OSLAR_EL1 is in the Core power domain

The OS Lock can also be locked or unlocked using [DBGOSLAR](#).

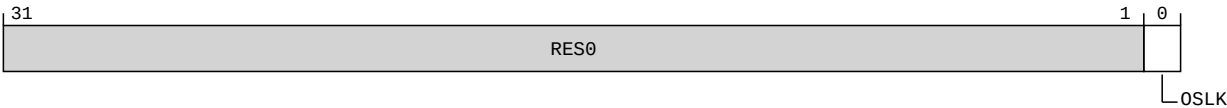
If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

If FEAT_Debugv8p2 is implemented, external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

Attributes

OSLAR_EL1 is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use [EDPRSR.OSLK](#) to check the current status of the lock.

Accessing OSLAR_EL1

OSLAR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x300	OSLAR_EL1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- When AllowExternalDebugAccess(addrdesc) and SoftwareLockStatus(addrdesc), accesses to this register are WI.
- When AllowExternalDebugAccess(addrdesc) and !SoftwareLockStatus(addrdesc), accesses to this register are WO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

E9.3 External trace registers

This section lists the external Trace unit registers.

E9.4 Cross-Trigger Interface registers

This section lists the Cross-Trigger Interface registers.

E9.4.1 ASICCTL, CTI External Multiplexer Control register

The ASICCTL characteristics are:

Purpose

Can be used to provide IMPLEMENTATION DEFINED controls for the CTI. For example, the register might be used to control multiplexors for additional IMPLEMENTATION DEFINED triggers. The IMPLEMENTATION DEFINED controls provided by this register might modify the architecturally defined behavior of the CTI.

Note
The architecturally-defined triggers must not be multiplexed.

Configuration

It is IMPLEMENTATION DEFINED whether ASICCTL is implemented in the Core power domain or in the Debug power domain

If it is implemented in the Core power domain, then it is IMPLEMENTATION DEFINED whether it is in the Cold reset domain or the Warm reset domain.

This register must reset to a value that supports the architecturally-defined behavior of the CTI. Changing the value of the register from its reset value causes IMPLEMENTATION DEFINED behavior that might differ from the architecturally-defined behavior of the CTI.

Other than the requirements listed in this register description, all aspects of the reset behavior of the ASICCTL are IMPLEMENTATION DEFINED.

Attributes

ASICCTL is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing ASICCTL

ASICCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x144	ASICCTL

Accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess(addrdesc), and SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

E9.4.2 CTIAPPCLEAR, CTI Application Trigger Clear register

The CTIAPPCLEAR characteristics are:

Purpose

Clears the application triggers.

Configuration

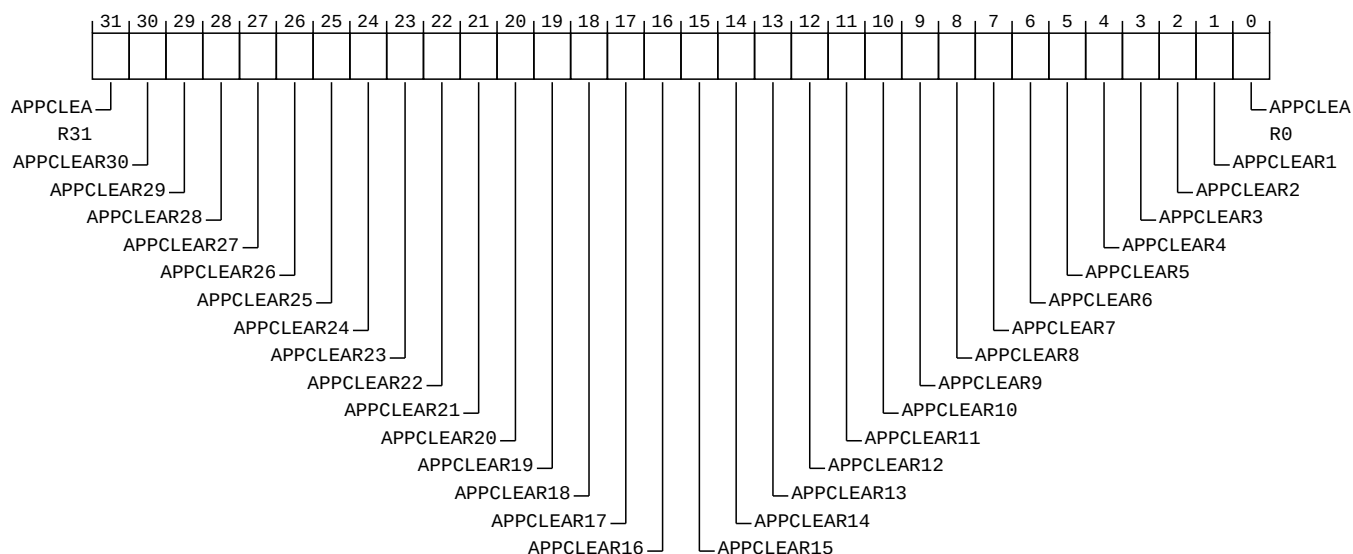
CTIAPPCLEAR is in the Debug power domain

External register CTIAPPCLEAR bits [31:0] are architecturally mapped to External register [CTIAPPSET](#)[31:0].

Attributes

CTIAPPCLEAR is a 32-bit register.

Field descriptions



APPCLEAR<x> , bits [x], for x = 31 to 0

Application trigger <x> disable.

Writing to this bit has the following effect:

APPCLEAR<x>	Meaning
0b0	No effect.
0b1	Clear corresponding application trigger to 0 and clear the corresponding channel event.

If the ECT does not support multicycle channel events, use of CTIAPPCLEAR is deprecated and the debugger must only use [CTIAPPULSE](#).

Accessing this field has the following behavior:

- When $x \geq \text{UInt}(\text{CTIDEVID.NUMCHAN})$, access to this field is RAZ/WI.
- Otherwise, access to this field is RAZ/WIC.

Accessing CTIAPPCLEAR

CTIAPPCLEAR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x018	CTIAPPCLEAR

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

E9.4.3 CTIAPPPULSE, CTI Application Pulse register

The CTIAPPPULSE characteristics are:

Purpose

Causes event pulses to be generated on ECT channels.

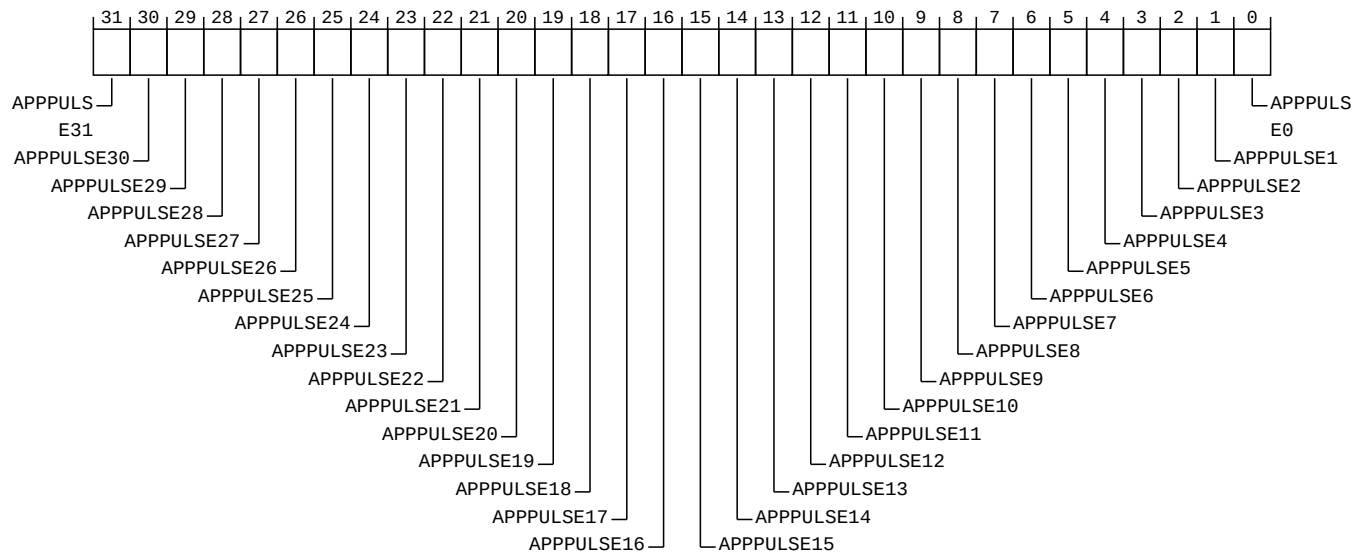
Configuration

CTIAPPPULSE is in the Debug power domain

Attributes

CTIAPPPULSE is a 32-bit register.

Field descriptions



APPPULSE<x> , bits [x], for x = 31 to 0

Generate event pulse on ECT channel <x>.

APPPULSE<x>	Meaning
0b0	No effect.
0b1	Channel <x> event pulse generated.

Note

- The CTIAPPPULSE operation does not affect the state of the application trigger. If the channel is active, either because of an earlier event or from the application trigger, then the value written to CTIAPPPULSE might have no effect.
- Multiple pulse events that occur close together might be merged into a single pulse event.

Accessing this field has the following behavior:

- When $x \geq \text{UInt}(\text{CTIDEVID.NUMCHAN})$, access to this field is RAZ/WI.
- Otherwise, access to this field is WO/UNKNOWN.

Accessing CTIAPPULSE

It is CONSTRAINED UNPREDICTABLE whether a write to CTIAPPULSE generates an event on a channel if CTICONTROL.GLBEN is 0.

CTIAPPULSE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x01C	CTIAPPULSE

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

E9.4.4 CTIAPPSET, CTI Application Trigger Set register

The CTIAPPSET characteristics are:

Purpose

Sets the application triggers.

Configuration

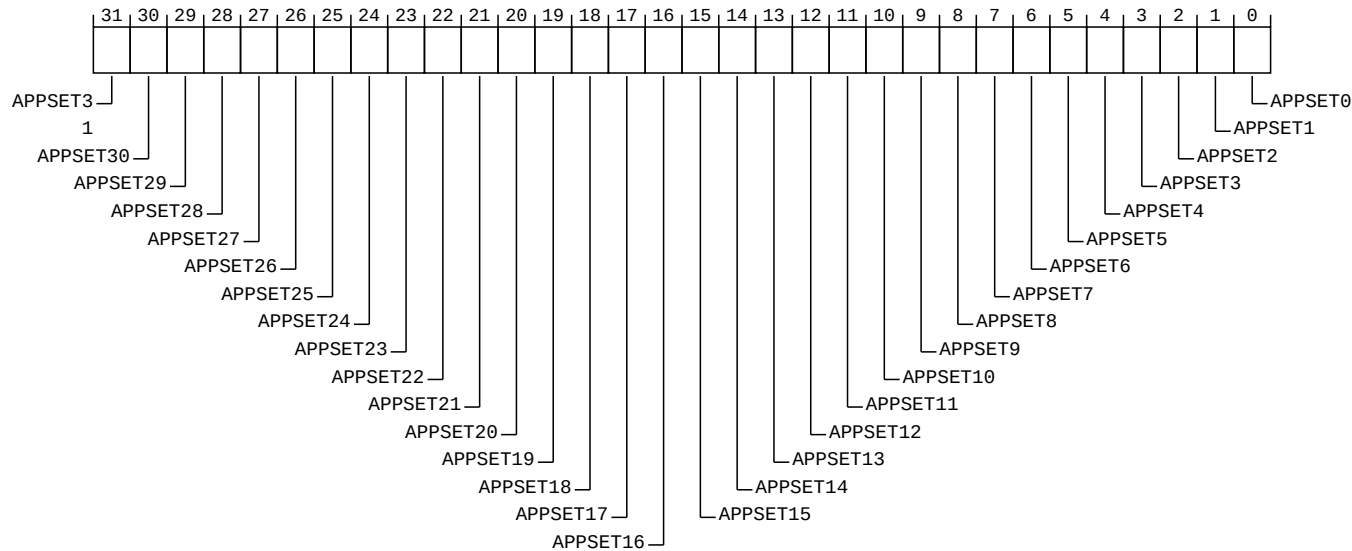
CTIAPPSET is in the Debug power domain

External register CTIAPPSET bits [31:0] are architecturally mapped to External register CTIAPPCLEAR[31:0].

Attributes

CTIAPPSET is a 32-bit register.

Field descriptions



APPSET<x> , bits [x], for x = 31 to 0

Application trigger <x> enable.

APPSET<x>	Meaning
0b0	Reading this means the application trigger is inactive. Writing this has no effect.
0b1	Reading this means the application trigger is active. Writing this sets the corresponding application trigger to 1 and generates a channel event.

If the ECT does not support multicycle channel events, use of CTIAPPSET is deprecated and the debugger must only use CTIAPPULSE.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $x \geq \text{UInt}(\text{CTIDEVID.NUMCHAN})$, access to this field is RAZ/WI.
- Otherwise, access to this field is WIS.

Accessing CTIAPPSET

CTIAPPSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0×014	CTIAPPSET

Accessible as follows:

- When `SoftwareLockStatus(addrdesc)`, accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.4.5 CTIAUTHSTATUS, CTI Authentication Status register

The CTIAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for CTI.

Configuration

CTIAUTHSTATUS is in the Debug power domain

This register is OPTIONAL, and is required for CoreSight compliance.

Attributes

CTIAUTHSTATUS is a 32-bit register.

Field descriptions

31	8	7	4	3	2	1	0
RES0							
RAZ				NSNID		NSID	

Bits [31:8]

Reserved, RES0.

Bits [7:4]

Reserved, RAZ.

NSNID, bits [3:2]

If the implemented Security state is Secure state, this field holds the same value as [DBGAUTHSTATUS_EL1.SNID](#).

NSID, bits [1:0]

If the implemented Security state is Secure state, this field holds the same value as [DBGAUTHSTATUS_EL1.SID](#).

Accessing CTIAUTHSTATUS

CTIAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFB8	CTIAUTHSTATUS

Accesses to this register are RO.

E9.4.6 CTICHINSTATUS, CTI Channel In Status register

The CTICHINSTATUS characteristics are:

Purpose

Provides the raw status of the ECT channel inputs to the CTI.

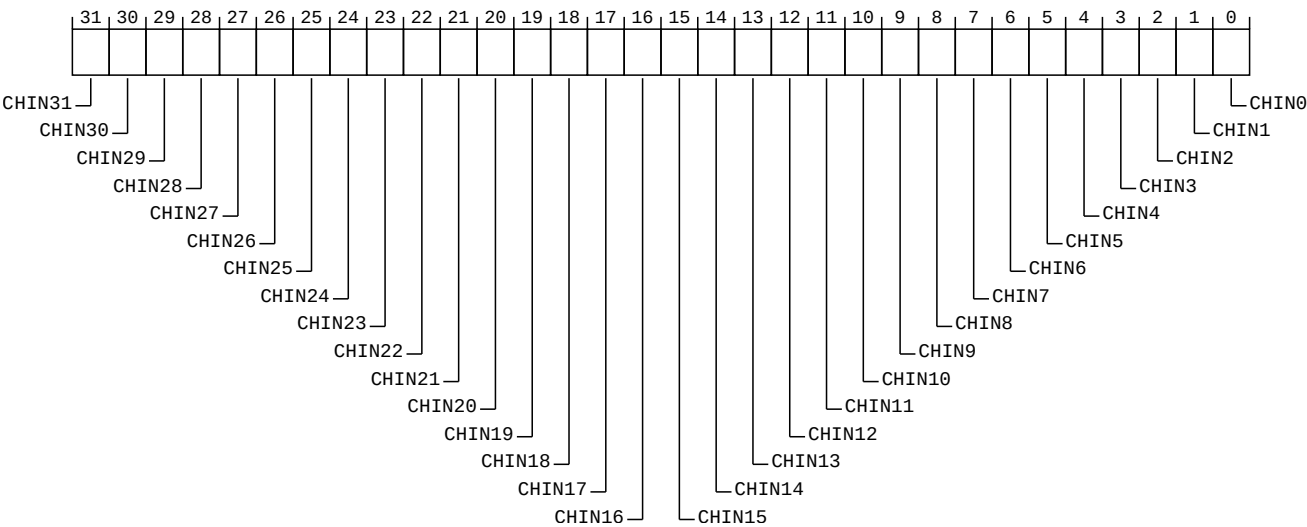
Configuration

CTICHINSTATUS is in the Debug power domain

Attributes

CTICHINSTATUS is a 32-bit register.

Field descriptions



CHIN<n> , bits [n], for n = 31 to 0

Input channel <n> status.

CHIN<n>	Meaning
0b0	Input channel <n> is inactive.
0b1	Input channel <n> is active.

If the ECT channels do not support multicycle events, then it is IMPLEMENTATION DEFINED whether an input channel can be observed as active.

Accessing this field has the following behavior:

- When $n \geq \text{UInt}(\text{CTIDEVID.NUMCHAN})$, access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Accessing CTICHINSTATUS

CTICHINSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x138	CTICHINSTATUS

Accesses to this register are RO.

E9.4.7 CTICHOUTSTATUS, CTI Channel Out Status register

The CTICHOUTSTATUS characteristics are:

Purpose

Provides the status of the ECT channel outputs from the CTI.

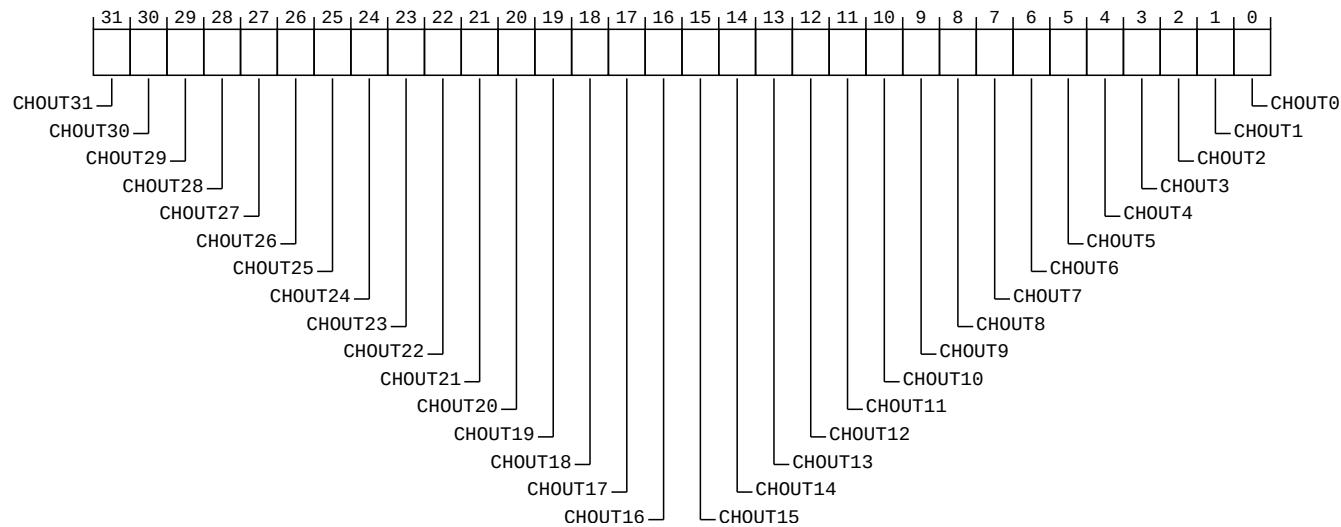
Configuration

CTICHOUTSTATUS is in the Debug power domain

Attributes

CTICHOUTSTATUS is a 32-bit register.

Field descriptions



CHOUT<n> , bits [n], for n = 31 to 0

Output channel <n> status.

CHOUT<n>	Meaning
0b0	Output channel <n> is inactive.
0b1	Output channel <n> is active.

If the ECT channels do not support multicycle events, then it is IMPLEMENTATION DEFINED whether an output channel can be observed as active.

Note
The value in CTICHOUTSTATUS is after gating by the channel gate. For more information, see [CTIGATE](#).

Accessing this field has the following behavior:

- When $n \geq \text{UInt}(\text{CTIDEVID.NUMCHAN})$, access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Accessing CTICHOUTSTATUS

CTICHOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x13C	CTICHOUTSTATUS

Accesses to this register are RO.

E9.4.8 CTICIDR0, CTI Component Identification Register 0

The CTICIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see ‘About the Component Identification scheme’.

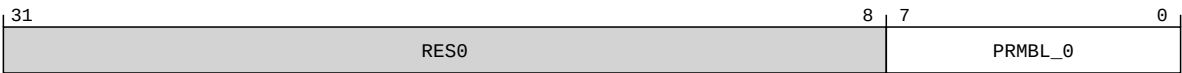
Configuration

CTICIDR0 is in the Debug power domain

Attributes

CTICIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

When FEAT_CTI_PIDR is implemented:

Preamble.
Reads as 0x0D
Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing CTICIDR0

CTICIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFF0	CTICIDR0

Accesses to this register are RO.

E9.4.9 CTICIDR1, CTI Component Identification Register 1

The CTICIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR1 is in the Debug power domain

Attributes

CTICIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

When FEAT_CTI_PIDR is implemented:

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

PRMBL_1, bits [3:0]

When FEAT_CTI_PIDR is implemented:

Preamble.

Reads as 0b0000

Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing CTICIDR1

CTICIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFF4	CTICIDR1

Accesses to this register are RO.

E9.4.10 CTICIDR2, CTI Component Identification Register 2

The CTICIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see ‘About the Component Identification scheme’.

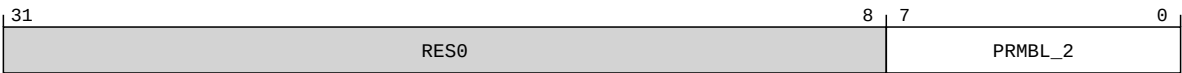
Configuration

CTICIDR2 is in the Debug power domain

Attributes

CTICIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

When FEAT_CTI_PIDR is implemented:

Preamble.
Reads as 0x05
Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing CTICIDR2

CTICIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFF8	CTICIDR2

Accesses to this register are RO.

E9.4.11 CTICIDR3, CTI Component Identification Register 3

The CTICIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see ‘About the Component Identification scheme’.

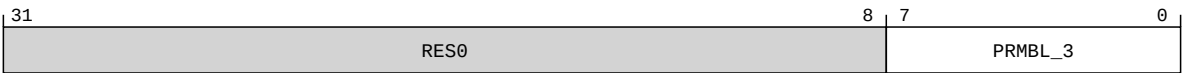
Configuration

CTICIDR3 is in the Debug power domain

Attributes

CTICIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

When FEAT_CTI_PIDR is implemented:

Preamble.
Reads as 0xB1
Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing CTICIDR3

CTICIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFFC	CTICIDR3

Accesses to this register are RO.

E9.4.12 CTICLAIMCLR, CTI CLAIM Tag Clear register

The CTICLAIMCLR characteristics are:

Purpose

Used by software to read the values of the CLAIM bits, and to clear CLAIM tag bits to 0.

Configuration

CTICLAIMCLR is in the Debug power domain

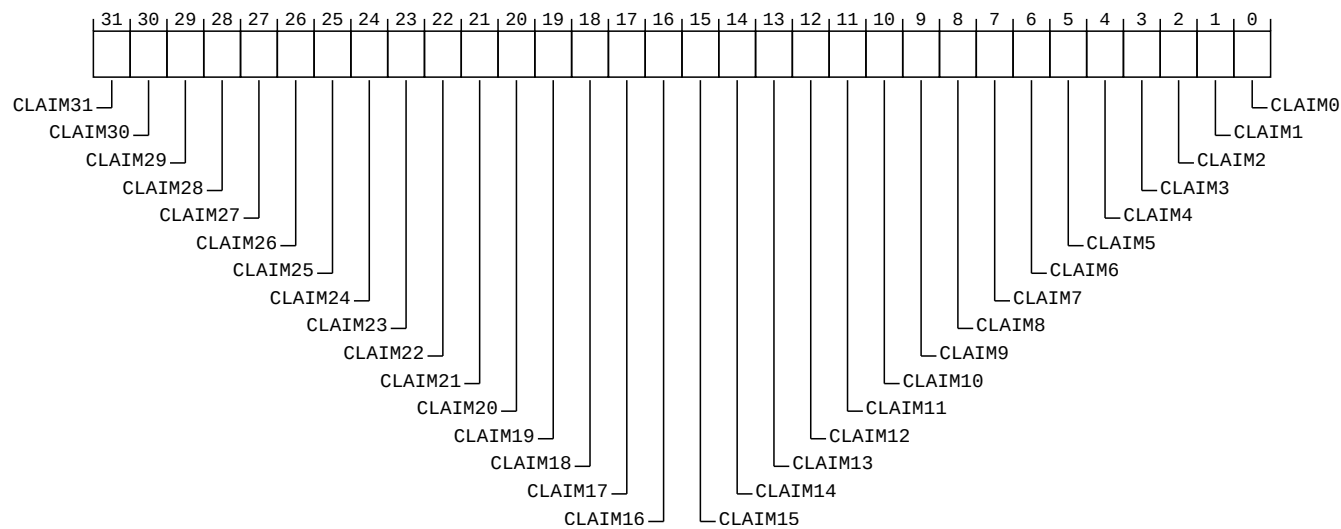
External register CTICLAIMCLR bits [31:0] are architecturally mapped to External register [CTICLAIMSET](#)[31:0].

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMCLR is a 32-bit register.

Field descriptions



CLAIM<m>, bits [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in [CTICLAIMSET](#).

An External Debug reset clears the CLAIM tag bits to 0.

Accessing this field has the following behavior:

- When $m \geq \text{NUM_CTI_CLAIM_TAGS}$, access to this field is RAZ/WI.
- Otherwise, access to this field is WIC.

Accessing CTICLAIMCLR

CTICLAIMCLR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFA4	CTICLAIMCLR

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.4.13 CTICLAIMSET, CTI CLAIM Tag Set register

The CTICLAIMSET characteristics are:

Purpose

Used by software to set CLAIM bits to 1.

Configuration

CTICLAIMSET is in the Debug power domain

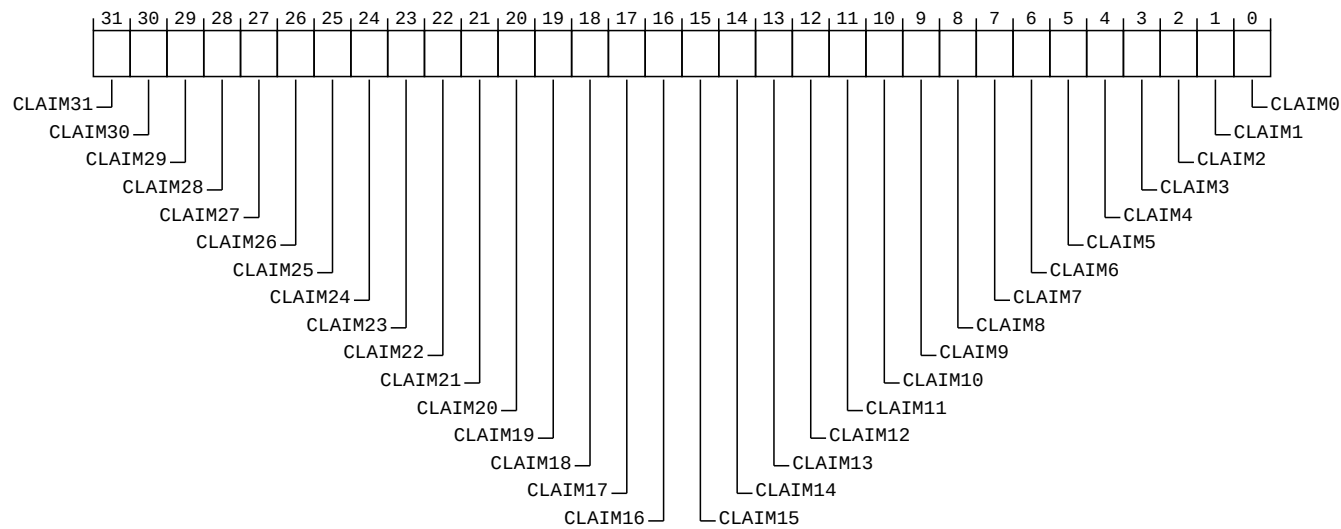
External register CTICLAIMSET bits [31:0] are architecturally mapped to External register [CTICLAIMCLR](#)[31:0].

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMSET is a 32-bit register.

Field descriptions



CLAIM<m> , bits [m], for m = 31 to 0

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not implemented. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is implemented. On a write: Set Claim Tag bit <m> to 1.

An External Debug reset clears the CLAIM tag bits to 0.

Accessing this field has the following behavior:

- When $m \geq \text{NUM_CTI_CLAIM_TAGS}$, access to this field is RAZ/WI.
- Otherwise, access to this field is RAO/WIS.

Accessing CTICLAIMSET

CTICLAIMSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFA0	CTICLAIMSET

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.4.14 CTICONTROL, CTI Control register

The CTICONTROL characteristics are:

Purpose

Controls whether the CTI is enabled.

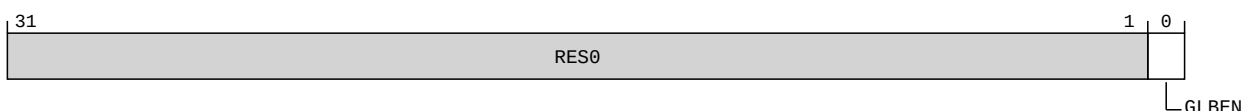
Configuration

CTICONTROL is in the Debug power domain

Attributes

CTICONTROL is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

GLBEN, bit [0]

Enables or disables the CTI mapping functions. Possible values of this field are:

GLBEN	Meaning
0b0	CTI mapping functions and application trigger disabled.
0b1	CTI mapping functions and application trigger enabled.

When GLBEN is 0, the input channel to output trigger, input trigger to output channel, and application trigger functions are disabled and do not signal new events on either output triggers or output channels. If a previously asserted output trigger has not been acknowledged, it is **CONSTRAINED UNPREDICTABLE** which of the following occurs:

- The output trigger remains asserted after the mapping functions are disabled.
- The output trigger is deasserted after the mapping functions are disabled.

All output triggers are disabled by CTI reset.

If the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Accessing CTICONTROL

CTICONTROL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x000	CTICONTROL

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.4.15 CTIDEVAFF0, CTI Device Affinity register 0

The CTIDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

Configuration

CTIDEVAFF0 is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

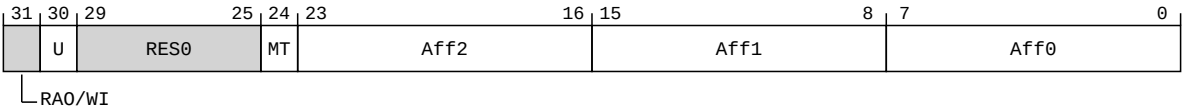
Arm recommends that the CTI is CTIv2.

If this register is implemented, then CTIDEVAFF1 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Attributes

CTIDEVAFF0 is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the [MPIDR](#).{Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVAFF0

CTIDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFA8	CTIDEVAFF0

Accesses to this register are RO.

E9.4.16 CTIDEVAFF1, CTI Device Affinity register 1

The CTIDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

Configuration

CTIDEVAFF1 is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

If this register is implemented, then CTIDEVAFF0 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Attributes

CTIDEVAFF1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of CTIDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVAFF1

CTIDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFAC	CTIDEVAFF1

Accesses to this register are RO.

E9.4.17 CTIDEVARCH, CTI Device Architecture register

The CTIDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the CTI component.

Configuration

CTIDEVARCH is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

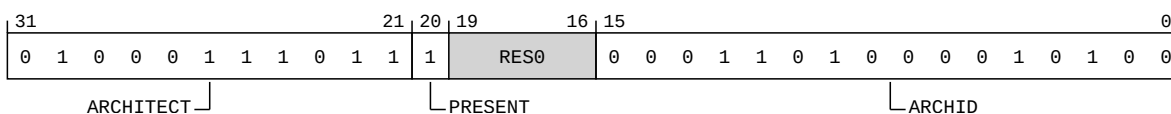
Arm recommends that the CTI is CTIv2.

If this register is not implemented, **CTIDEVAFF0** and **CTIDEVAFF1** are also not implemented.

Attributes

CTIDEVARCH is a 32-bit register.

Field descriptions

**ARCHITECT, bits [31:21]**

Defines the architect of the component. For CTI, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, 0b0100.

Bits [27:21] are the JEP106 identification code, 0b0111011.

Reads as 0b01000111011

Access to this field is RO.

PRESENT, bit [20]

DEVARCH presence. Indicates that the CTIDEVARCH register provides architecture information.

Reads as 0b1

Access to this field is RO.

Bits [19:16]

Reserved, RES0.

ARCHID, bits [15:0]

Defines this part to be an Armv8 debug component. For architectures defined by Arm this is further subdivided.

For CTI:

- Bits [15:12] are the architecture version, 0x1.
- Bits [11:0] are the architecture part number, 0xA14.

This corresponds to CTI architecture version CTIv2.

Reads as 0x1A14

Access to this field is RO.

Accessing CTIDEVARCH

CTIDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFBC	CTIDEVARCH

Accesses to this register are RO.

E9.4.18 CTIDEVID, CTI Device ID register 0

The CTIDEVID characteristics are:

Purpose

Describes the CTI component to the debugger.

Configuration

CTIDEVID is in the Debug power domain

Attributes

CTIDEVID is a 32-bit register.

Field descriptions

31	26	25	24	23	22	21	16	15	14	13	8	7	5	4	0
RES0				INOUT	RES0	NUMCHAN				RES0	NUMTRIG			RES0	EXTMUXNUM

Bits [31:26]

Reserved, RES0.

INOUT, bits [25:24]

Input/output options. Indicates presence of the input gate. If the CTM is not implemented or CTIv2 is not implemented, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

INOUT	Meaning
0b00	CTIGATE does not mask propagation of input events from external channels.
0b01	CTIGATE masks propagation of input events from external channels.

All other values are reserved.

Access to this field is RO.

Bits [23:22]

Reserved, RES0.

NUMCHAN, bits [21:16]

Number of ECT channels implemented. For Armv8, valid values are:

- 0b000011 3 channels (0..2) implemented.
- 0b000100 4 channels (0..3) implemented.
- 0b000101 5 channels (0..4) implemented.
- 0b000110 6 channels (0..5) implemented.

and so on up to 0b100000, 32 channels (0..31) implemented.

All other values are reserved.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [15:14]

Reserved, RES0.

NUMTRIG, bits [13:8]

Upper bound for number of triggers. The indices of all implemented input and output triggers are less than this value.

There is no guarantee that all of the input and output triggers, including the highest numbered, are connected to any components, or that the implementation of input and output triggers is symmetrical.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [7:5]

Reserved, RES0.

EXTMUXNUM, bits [4:0]

Number of multiplexors available on triggers. This value is used in conjunction with External Control register, [ASICCTL](#).

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVID

CTIDEVID can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC8	CTIDEVID

Accesses to this register are RO.

E9.4.19 CTIDEVID1, CTI Device ID register 1

The CTIDEVID1 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

Configuration

CTIDEVID1 is in the Debug power domain

Attributes

CTIDEVID1 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing CTIDEVID1

CTIDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC4	CTIDEVID1

Accesses to this register are RO.

E9.4.20 CTIDEVID2, CTI Device ID register 2

The CTIDEVID2 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

Configuration

CTIDEVID2 is in the Debug power domain

Attributes

CTIDEVID2 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing CTIDEVID2

CTIDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC0	CTIDEVID2

Accesses to this register are RO.

E9.4.21 CTIDEVTYPE, CTI Device Type register

The CTIDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE’s cross-trigger interface.

Configuration

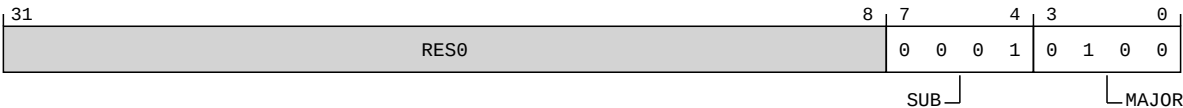
CTIDEVTYPE is in the Debug power domain

This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIDEVTYPE are RES0.

Attributes

CTIDEVTYPE is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as 0b0001

Access to this field is RO.

MAJOR, bits [3:0]

Major type. Indicates this is a cross-trigger component.

Reads as 0b0100

Access to this field is RO.

Accessing CTIDEVTYPE

CTIDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFCC	CTIDEVTYPE

Accesses to this register are RO.

E9.4.22 CTIGATE, CTI Channel Gate Enable register

The CTIGATE characteristics are:

Purpose

Determines whether events on channels propagate through the CTM to other ECT components, or from the CTM into the CTI.

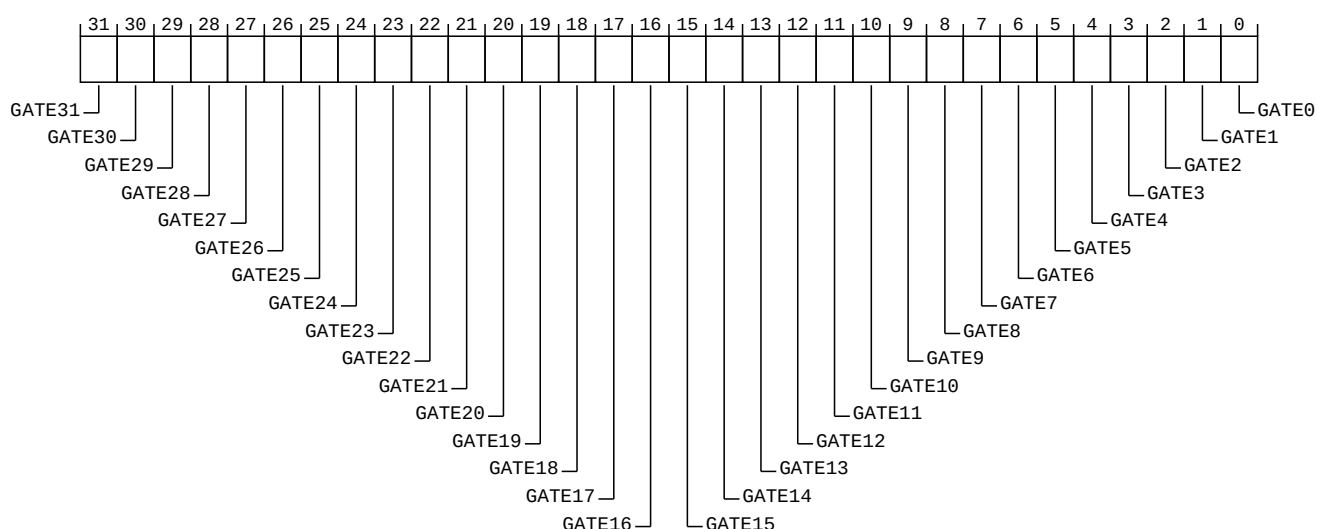
Configuration

CTIGATE is in the Debug power domain

Attributes

CTIGATE is a 32-bit register.

Field descriptions



GATE<x> , bits [x], for x = 31 to 0

Channel <x> gate enable.

GATE<x>	Meaning
0b0	Disable output and, if CTIDEVID.INOUT == 0b01, input channel <x> propagation.
0b1	Enable output and, if CTIDEVID.INOUT == 0b01, input channel <x> propagation.

If GATE<x> is set to 0, no new events will be propagated to the ECT, and if the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $x \geq \text{UInt}(\text{CTIDEVID.NUMCHAN})$, access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Accessing CTIGATE

CTIGATE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x140	CTIGATE

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.4.23 CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31

The CTIINEN<n> characteristics are:

Purpose

Enables the signaling of an event on output channels when input trigger event n is received by the CTI.

Configuration

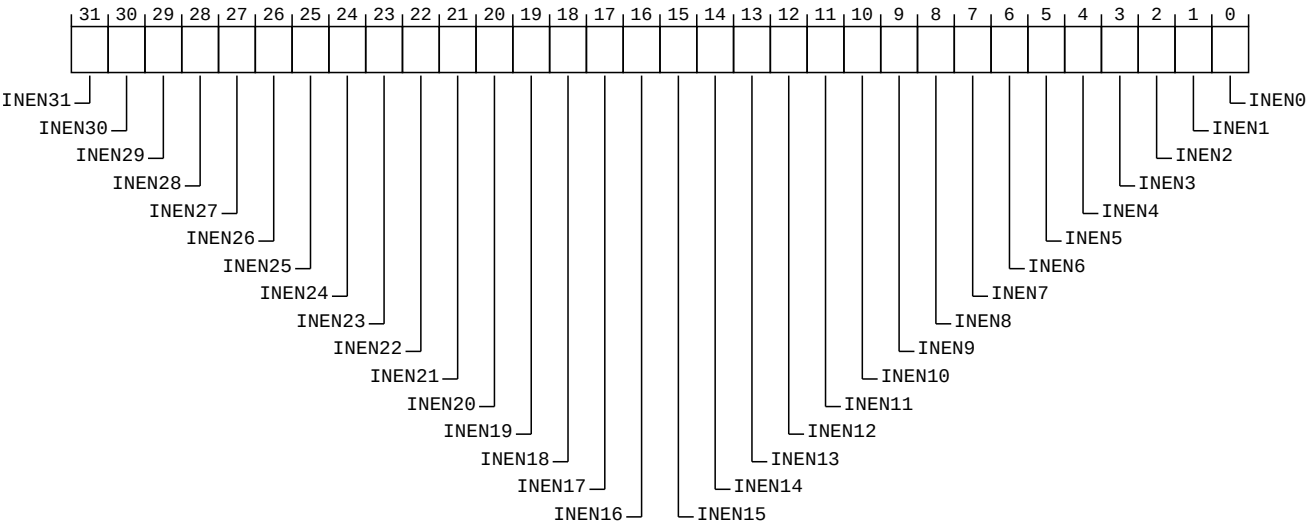
CTIINEN<n> is in the Debug power domain

If input trigger n is not implemented or not connected, CTIINEN<n> is RES0.

Attributes

CTIINEN<n> is a 32-bit register.

Field descriptions



INEN<x> , bits [x], for x = 31 to 0

Input trigger <n> to output channel <x> enable.

INEN<x>	Meaning
0b0	Input trigger <n> will not generate an event on output channel <x>.
0b1	Input trigger <n> will generate an event on output channel <x>.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Accessing CTIINEN<n>

CTIINEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	$0x020 + (4 * n)$	CTIINEN<n>

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.4.24 CTIINTACK, CTI Output Trigger Acknowledge register

The CTIINTACK characteristics are:

Purpose

Can be used to deactivate the output triggers.

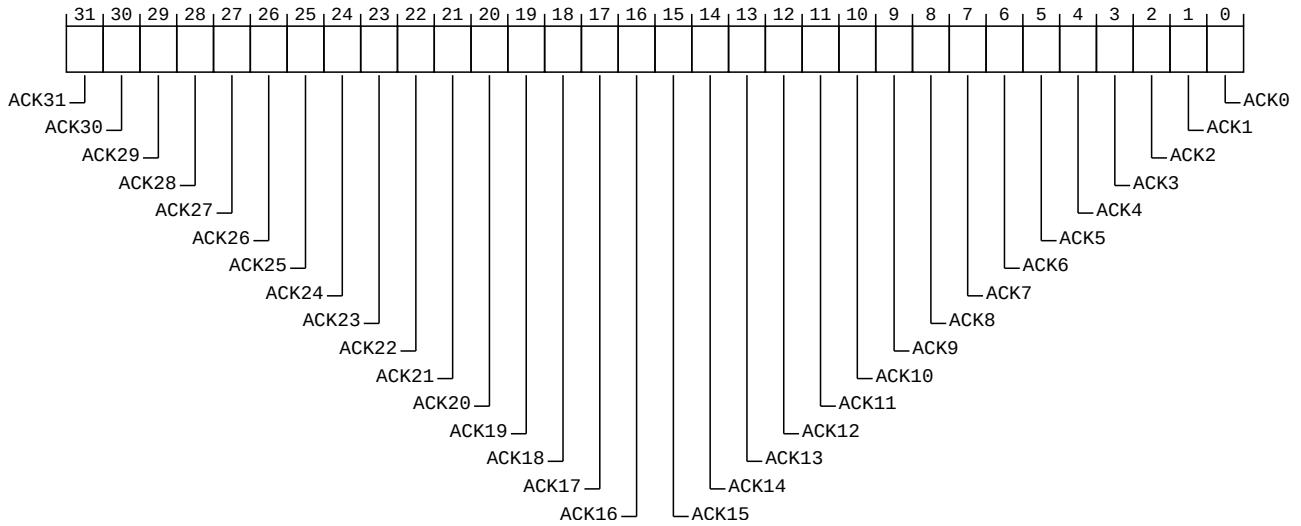
Configuration

CTIINTACK is in the Debug power domain

Attributes

CTIINTACK is a 32-bit register.

Field descriptions



ACK<n>, bits [n], for n = 31 to 0

Acknowledge for output trigger <n>.

If any of the following is true, writes to ACK<n> are ignored:

- $n \geq \text{CTIDEVID.NUMTRIG}$, the number of implemented triggers.
- Output trigger n is not active.
- The channel mapping function output, as controlled by [CTIOUTEN<n>](#), is still active.

Otherwise, if any of the following are true, ACK<n> is RES0:

- Output trigger n is not implemented.
- Output trigger n is not connected.
- Output trigger n is self-acknowledging and does not require software acknowledge.

Otherwise, the behavior on writes to ACK<n> is as follows:

ACK<n>	Meaning
0b0	No effect
0b1	Deactivate the trigger.

Accessing this field has the following behavior:

- When $n \geq \text{UInt}(\text{CTIDEVID.NUMTRIG})$, access to this field is RAZ/WI.
- Otherwise, access to this field is WO/UNKNOWN.

Accessing CTIINTACK

A debugger must read [CTITRIGOUTSTATUS](#) to confirm that the output trigger has been acknowledged before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPULSE to activate another trigger.

CTIINTACK can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x010	CTIINTACK

Accessible as follows:

- When `SoftwareLockStatus(addrdesc)`, accesses to this register are WI.
- Otherwise, accesses to this register are WO.

E9.4.25 CTIITCTRL, CTI Integration mode Control register

The CTIITCTRL characteristics are:

Purpose

Enables the CTI to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

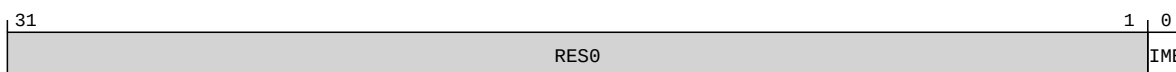
The power domain of CTIITCTRL is IMPLEMENTATION DEFINED

Implementation of this register is OPTIONAL.

Attributes

CTIITCTRL is a 32-bit register.

Field descriptions

**Bits [31:1]**

Reserved, RES0.

IME, bit [0]

Integration mode enable. When $\text{IME} = 1$, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
 - On a Cold reset, this field resets to 0.
 - On an External debug reset, the value of this field is unchanged.
 - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
 - On a Cold reset, the value of this field is unchanged.
 - On an External debug reset, this field resets to 0.
 - On a Warm reset, the value of this field is unchanged.

Accessing CTIITCTRL

CTIITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xF00	CTIITCTRL

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.4.26 CTILAR, CTI Lock Access Register

The CTILAR characteristics are:

Purpose

Allows or disallows access to the CTI registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

CTILAR is in the Debug power domain

If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

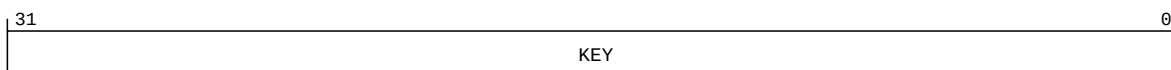
Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILAR is a 32-bit register.

Field descriptions

When CTI Software Lock is implemented:

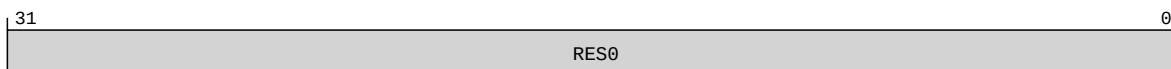


KEY, bits [31:0]

Lock Access control. Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:



Otherwise

Bits [31:0]

Reserved, RES0.

Accessing CTILAR

CTILAR can be accessed through the memory-mapped interface:

Component	Offset	Instance
CTI	0xFB0	CTILAR

Accesses to this register are WO.

E9.4.27 CTILSR, CTI Lock Status Register

The CTILSR characteristics are:

Purpose

Indicates the current status of the Software Lock for CTI registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

CTILSR is in the Debug power domain

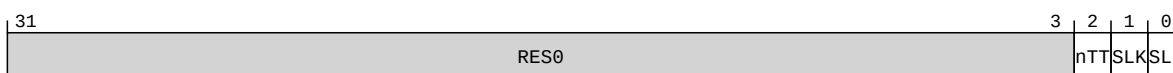
If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

Software uses **CTILAR** to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILSR is a 32-bit register.

Field descriptions

**Bits [31:3]**

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLK, bit [1]

When CTI Software Lock is implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when the Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when the Software Lock is implemented, possible values of this field are:

SLK	Meaning
0b0	Lock clear. Writes are permitted to this component's registers.
0b1	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a External debug reset, this field resets to '1'.

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
0b0	Software Lock not implemented or not memory-mapped access.
0b1	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing CTILSR

CTILSR can be accessed through the memory-mapped interface:

Component	Offset	Instance
CTI	0xFB4	CTILSR

Accesses to this register are RO.

E9.4.28 CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31

The CTIOUTEN<n> characteristics are:

Purpose

Defines which input channels generate output trigger n.

Configuration

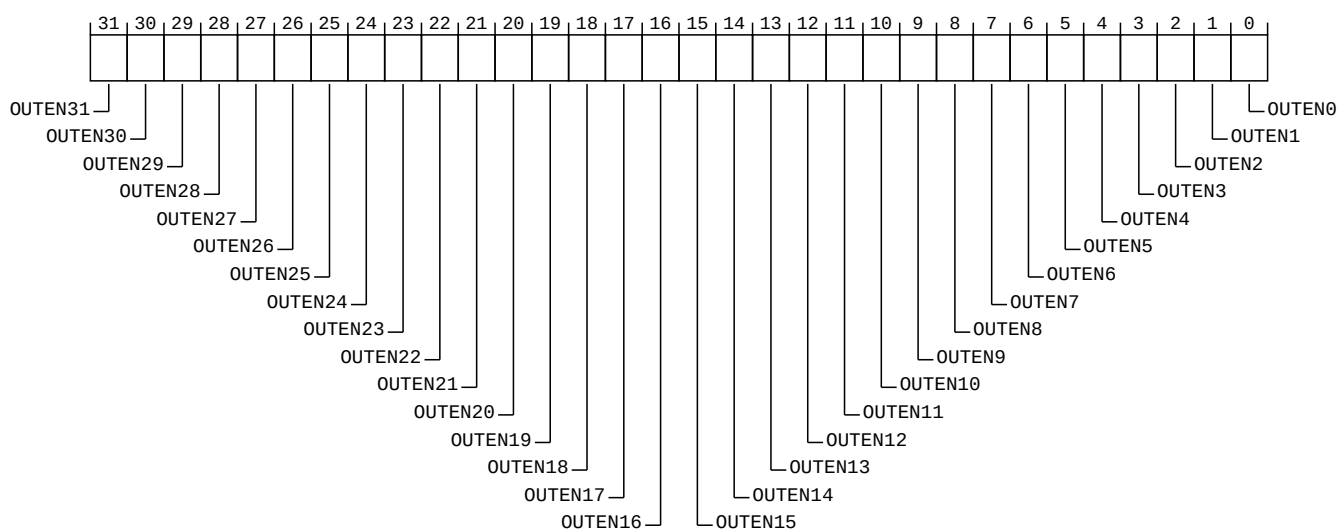
CTIOUTEN<n> is in the Debug power domain

If output trigger n is not implemented or not connected, CTIOUTEN<n> is RES0.

Attributes

CTIOUTEN<n> is a 32-bit register.

Field descriptions



OUTEN<x>, bits [x], for x = 31 to 0

Input channel <x> to output trigger <n> enable.

OUTEN<x>	Meaning
0b0	An event on input channel <x> will not cause output trigger <n> to be asserted.
0b1	An event on input channel <x> will cause output trigger <n> to be asserted.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $x \geq \text{UInt}(\text{CTIDEVID.NUMCHAN})$, access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Accessing CTIOUTEN<n>

CTIOUTEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	$0x0A0 + (4 * n)$	CTIOUTEN<n>

Accessible as follows:

- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

E9.4.29 CTIPIDR0, CTI Peripheral Identification Register 0

The CTIPIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see ‘About the Peripheral identification scheme’.

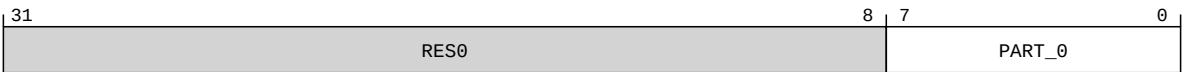
Configuration

CTIPIDR0 is in the Debug power domain
This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIPIDR0 are RES0.

Attributes

CTIPIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
 - CTIPIDR1.PART_1 contains part number bits [11:8].
 - CTIPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
 - CTIPIDR1.PART_1 contains part number bits [15:12].
 - CTIPIDR0.PART_0 contains part number bits [11:4].
 - CTIPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, CTIPIDR2.REVISION indicates revision information.
The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.
This field has an IMPLEMENTATION DEFINED value.
Access to this field is RO.

Accessing CTIPIDR0

CTIPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFE0	CTIPIDR0

Accesses to this register are RO.

E9.4.30 CTIPIDR1, CTI Peripheral Identification Register 1

The CTIPIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR1 is in the Debug power domain

This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIPIDR1 are RES0.

Attributes

CTIPIDR1 is a 32-bit register.

Field descriptions

31		8	7	4	3	0	
RES0						DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- **CTIPIDR1**.DES_0: JEP106 identification code bits[3:0].
- **CTIPIDR2**.DES_1: JEP106 identification code bits[6:4].
- **CTIPIDR4**.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value 0x7F.
- A following 8-bit number, that is not 0x7F, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code 0x7F 0x7F 0x7F 0x7F 0x3B.

- The continuation code is the number of times 0x7F appears before the final number. For example, a component designed by Arm Limited has the code 0x4.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code 0x3B.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
 - CTIPIDR1.PART_1 contains part number bits [11:8].
 - CTIPIDR0.PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
 - CTIPIDR1.PART_1 contains part number bits [15:12].
 - CTIPIDR0.PART_0 contains part number bits [11:4].
 - CTIPIDR2.REVISION contains part number bits [3:0].

When a 12-bit part number is used, CTIPIDR2.REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR1

CTIPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFE4	CTIPIDR1

Accesses to this register are RO.

E9.4.31 CTIPIDR2, CTI Peripheral Identification Register 2

The CTIPIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

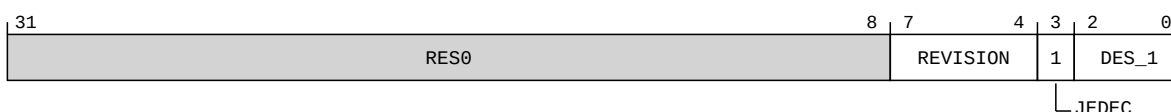
CTIPIDR2 is in the Debug power domain

This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIPIDR2 are RES0.

Attributes

CTIPIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- [CTIPIDR2.REVISION](#) indicates the 4-bit revision number.
- [CTIPIDR3.REVAND](#) indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- [CTIPIDR2.REVISION](#) indicates the 4-bit major revision number.
- [CTIPIDR3.REVAND](#) indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- [CTIPIDR2.REVISION](#) contains part number bits [3:0].
- [CTIPIDR3.REVAND](#) indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as 0b1

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- CTIPIDR1.DES_0: JEP106 identification code bits[3:0].
- CTIPIDR2.DES_1: JEP106 identification code bits[6:4].
- CTIPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value 0x7F.
- A following 8-bit number, that is not 0x7F, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code 0x7F 0x7F 0x7F 0x7F 0x3B.

- The continuation code is the number of times 0x7F appears before the final number. For example, a component designed by Arm Limited has the code 0x4.
 - The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code 0x3B.
-

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR2

CTIPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFE8	CTIPIDR2

Accesses to this register are RO.

E9.4.32 CTIPIDR3, CTI Peripheral Identification Register 3

The CTIPIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR3 is in the Debug power domain

This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIPIDR3 are RES0.

Attributes

CTIPIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- [CTIPIDR2.REVISION](#) indicates the 4-bit revision number.
- [CTIPIDR3.REVAND](#) indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- [CTIPIDR2.REVISION](#) indicates the 4-bit major revision number.
- [CTIPIDR3.REVAND](#) indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- [CTIPIDR2.REVISION](#) contains part number bits [3:0].
- [CTIPIDR3.REVAND](#) indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field

is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

CTIPIDR3.CMOD might also indicate component modifications.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of 0b0000 means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

CTIPIDR3.REVAND might also indicate component modifications.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR3

CTIPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFEC	CTIPIDR3

Accesses to this register are RO.

E9.4.33 CTIPIDR4, CTI Peripheral Identification Register 4

The CTIPIDR4 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

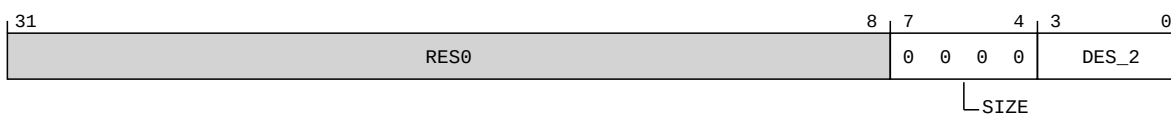
CTIPIDR4 is in the Debug power domain

This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIPIDR4 are RES0.

Attributes

CTIPIDR4 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. $\log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as 0b0000

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

- CTIPIDR1.DES_0: JEP106 identification code bits[3:0].
- CTIPIDR2.DES_1: JEP106 identification code bits[6:4].
- CTIPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value 0x7F.
- A following 8-bit number, that is not 0x7F, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code 0x7F 0x7F 0x7F 0x7F 0x3B.

- The continuation code is the number of times 0x7F appears before the final number. For example, a component designed by Arm Limited has the code 0x4.
 - The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code 0x3B.
-

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR4

CTIPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFD0	CTIPIDR4

Accesses to this register are RO.

E9.4.34 CTIPIDR5, CTI Peripheral Identification Register 5

The CTIPIDR5 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see ‘About the Peripheral identification scheme’.

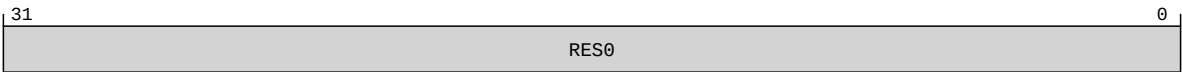
Configuration

CTIPIDR5 is in the Debug power domain
This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIPIDR5 are RES0.

Attributes

CTIPIDR5 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing CTIPIDR5

CTIPIDR5 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFD4	CTIPIDR5

Accesses to this register are RO.

E9.4.35 CTIPIDR6, CTI Peripheral Identification Register 6

The CTIPIDR6 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see ‘About the Peripheral identification scheme’.

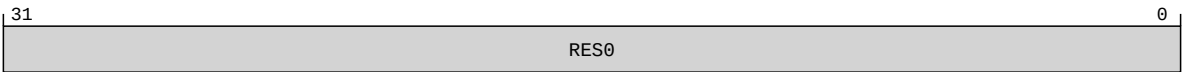
Configuration

CTIPIDR6 is in the Debug power domain
This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIPIDR6 are RES0.

Attributes

CTIPIDR6 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing CTIPIDR6

CTIPIDR6 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFD8	CTIPIDR6

Accesses to this register are RO.

E9.4.36 CTIPIDR7, CTI Peripheral Identification Register 7

The CTIPIDR7 characteristics are:

Purpose

Provides information to identify a CTI component.
For more information, see ‘About the Peripheral identification scheme’.

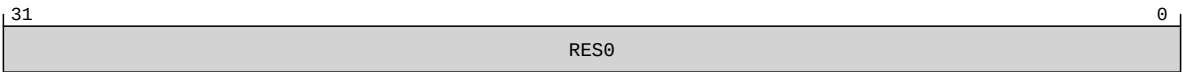
Configuration

CTIPIDR7 is in the Debug power domain
This register is present only when FEAT_CTI_PIDR is implemented. Otherwise, direct accesses to CTIPIDR7 are RES0.

Attributes

CTIPIDR7 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing CTIPIDR7

CTIPIDR7 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFDC	CTIPIDR7

Accesses to this register are RO.

E9.4.37 CTITRIGINSTATUS, CTI Trigger In Status register

The CTITRIGINSTATUS characteristics are:

Purpose

Provides the status of the trigger inputs.

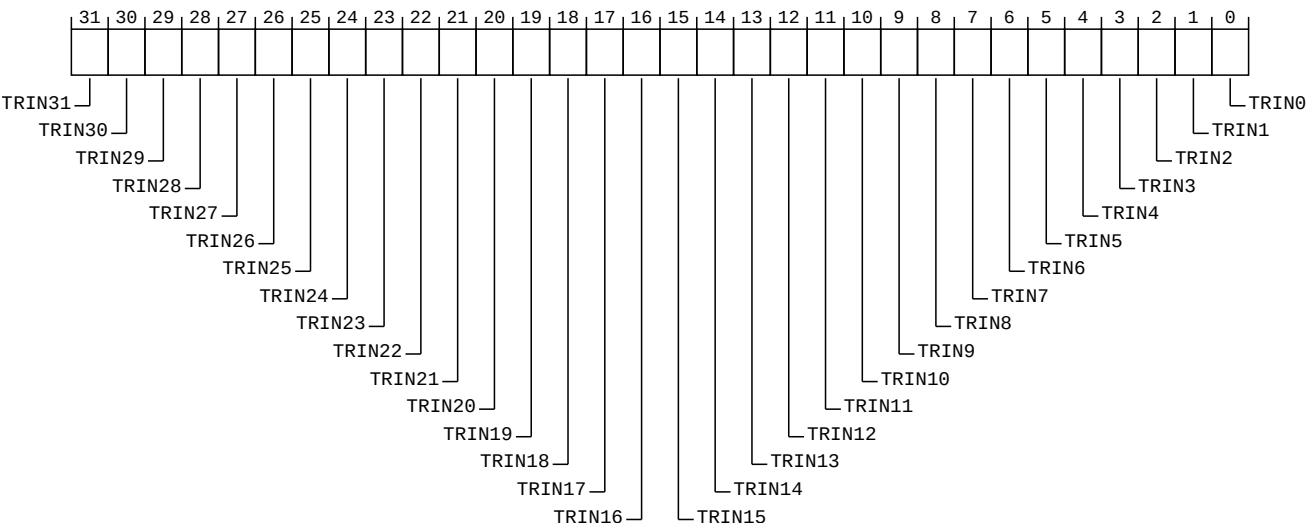
Configuration

CTITRIGINSTATUS is in the Debug power domain

Attributes

CTITRIGINSTATUS is a 32-bit register.

Field descriptions



TRIN<n> , bits [n], for n = 31 to 0

Trigger input <n> status.

TRIN<n>	Meaning
0b0	Input trigger n is inactive.
0b1	Input trigger n is active.

Not implemented and not-connected input triggers are always inactive.

It is IMPLEMENTATION DEFINED whether an input trigger that does not support multicyle events can be observed as active.

Accessing this field has the following behavior:

- When $n \geq \text{UInt}(\text{CTIDEVID.NUMTRIG})$, access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Accessing CTITRIGINSTATUS

CTITRIGINSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x130	CTITRIGINSTATUS

Accesses to this register are RO.

E9.4.38 CTITRIGOUTSTATUS, CTI Trigger Out Status register

The CTITRIGOUTSTATUS characteristics are:

Purpose

Provides the raw status of the trigger outputs, after processing by any IMPLEMENTATION DEFINED trigger interface logic. For output triggers that are self-acknowledging, this is only meaningful if the CTI implements multicycle channel events.

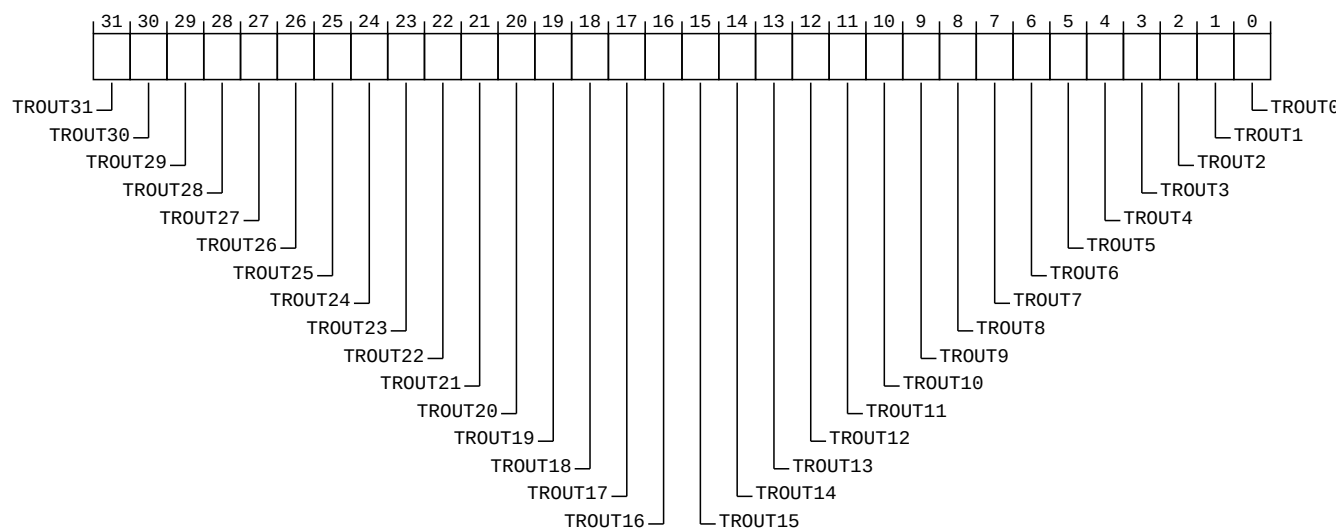
Configuration

CTITRIGOUTSTATUS is in the Debug power domain

Attributes

CTITRIGOUTSTATUS is a 32-bit register.

Field descriptions



TROUT<n> , bits [n], for n = 31 to 0

Trigger output <n> status.

If n is less than CTIDEVID.NUMTRIG, and output trigger <n> is implemented and connected, and either the trigger is not self-acknowledging or the CTI implements multicycle channel events, then permitted values for TROUT<n> are:

TROUT<n>	Meaning
0b0	Output trigger n is inactive.
0b1	Output trigger n is active.

Otherwise when n is less than CTIDEVID.NUMTRIG, it is IMPLEMENTATION DEFINED whether TROUT<n> behaves as described here or is RAZ.

Accessing this field has the following behavior:

- When $n \geq \text{UInt}(\text{CTIDEVID.NUMTRIG})$, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
 - Any of the following are true:
 - Output trigger $\langle n \rangle$ is not implemented or not connected
 - All of the following are true:
 - Output trigger $\langle n \rangle$ is self-acknowledging
 - CTI does not implement multicycle channel events
 - an implementation implements $\text{TROUT}\langle n \rangle$
- Otherwise, access to this field is RO.

Accessing CTITRIGOUTSTATUS

CTITRIGOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0×134	CTITRIGOUTSTATUS

Accesses to this register are RO.

Part F
**Memory-mapped Components of the Arm
Architecture**

Chapter F1

Requirements for Memory-mapped Components

This chapter provides some additional information about memory-mapped components defined by the Armv8 and Armv9 architectures. Arm also recommends that memory-mapped components designed to support the Armv8 and Armv9 architectures use this section as guidance, other than where a conflicting behavior is required by some other standard that the component is designed to support. It contains the following sections:

- [Supported access sizes and Endianness](#)
- [Synchronization of memory-mapped registers](#)
- [Access requirements for reserved and unallocated registers](#)

F1.1 Supported access sizes and Endianness

The information in this section applies to all accesses to memory-mapped components of the Armv8 and later architectures, unless a register or component description explicitly states otherwise.

The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral.

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

An implementation of a memory-mapped component that is compatible with the Armv8 and later architectures must support the following:

- Word-aligned 32-bit accesses to access 32-bit registers.
- If the system includes any component with direct memory access to the memory-mapped component which needs to access the component but does not support making 64-bit accesses, word-aligned 32-bit accesses to either half of a 64-bit register. This includes, but is not limited to:
 - A PE that implements a 32-bit ISA that does not include 64-bit memory operations. For example, Armv8-M T32.
 - A Debug Access Port that cannot make 64-bit accesses.
 - A PE that supports AArch32 at any Exception level.Arm deprecates support for 32-bit accesses to either half of 64-bit registers.

Note

Although AArch32 implementations might make 64-bit accesses for LDP and STP instructions, this is not architecturally required. To guarantee ordering of accesses, portable AArch32 software should use LDR and STR. For compatibility with such software, the memory-mapped component should treat the PE that supports AArch32 as not making 64-bit accesses using portable code.

- Doubleword-aligned 64-bit accesses to access 64-bit registers that are mapped to a doubleword-aligned pair of adjacent 32-bit locations.

Unless otherwise specified, all registers are only single-copy atomic at word granularity. This means that for 64-bit accesses to a 64-bit register, the system might generate a pair of 32-bit accesses. The order in which the two halves are accessed is not specified.

The following accesses are not supported:

- Byte accesses.
- Halfword accesses.
- Unaligned word accesses. These accesses are not word single-copy atomic.
- Unaligned doubleword accesses. These accesses are not doubleword single-copy atomic.
- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair that forms a 64-bit register.
- Quadword accesses or higher accesses.
- Exclusive accesses.

For unsupported accesses, it is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether:

- The access generates an External abort or not.
- The defined side-effects of a read occur or not. A read returns an IMPLEMENTATION DEFINED value that might be UNKNOWN.
- A write is ignored or sets the accessed register or registers to UNKNOWN values.
- The access generates a fault handling interrupt or not.

Such accesses are often due to software faults. That is, software has accessed a location in a way that the component does not support.

For memory-mapped accesses from a PE that complies with an Arm architecture, the single-copy atomicity rules for the instruction, the type of instruction, and the type of memory that is accessed, determine the size of the access that is made by an instruction. [Example F1-1](#) shows this.

Example F1-1 Access sizes for memory-mapped accesses

Two Load Doubleword instructions that are made to consecutive doubleword-aligned locations generate a pair of single-copy atomic doubleword reads. However, if the accesses are made to Normal memory or Device-GRE memory they might appear as a single quadword access that is not supported by the peripheral.

The Arm architecture does not require the size of each element that is accessed by a multi-register load or store instruction to be identifiable by the memory system beyond the PE. Unless otherwise specified by the component, any access to a memory-mapped component of the Arm architecture is defined to be beyond the PE.

Software must use a Device-nGRE or stronger memory type, and only single register load and store instructions, to create memory accesses that are supported by the peripheral. For more information, see Memory types and attributes.

F1.2 Synchronization of memory-mapped registers

This section describes the synchronization requirements for the memory-mapped accesses to System registers.

This section refers to accesses to external system control registers as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

[Synchronization of changes to AArch32 System registers](#) define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads and external writes as indirect writes.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. Unless otherwise stated, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, and so is never required for registers that are accessible as external system control registers.

Unless required to be observable to all observers in finite time, without explicit synchronization, explicit synchronization is normally required following an external write to any register for that write to be observable by:

- A direct access.
- An indirect read by an instruction.
- An external read of the register using a different address.

This means that an external write by an external agent is guaranteed to have an effect on subsequent instructions executed by the PE only if all of the following are true:

- The write has completed.
- The PE has executed a Context synchronization event.
- The Context synchronization event was executed after the write completed.

The order and synchronization of direct reads and direct writes of System registers is defined by [Synchronization of changes to AArch32 System registers](#).

The external agent must be able to guarantee completion of a write. For example, the agent can:

- Mark the memory as Device-nGnRnE and executing a DSB barrier, if the system supports this property.
- If the register is read/write and reads are not destructive, read back the value written.
- Use some guaranteed property of the connection between the PE and the external agent.

The external agent and PE can guarantee ordering by, for example, passing messages in an ordered way with respect to the external write and the Context synchronization event, and relying on the memory ordering rules provided by the memory model.

External reads and external write complete in the order in which they arrive at the PE. For accesses to different register locations, the external agent must create this order. The agent can:

- Mark the memory as Device-nGnRnE or Device-nGnRE.
- Use the appropriate memory barriers.
- Rely on some guaranteed property of the connection between the PE and the external agent.

However, the external agent cannot force the synchronization of completed writes.

In a simple sequential execution, an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a

memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

For example, some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means the simultaneous writes must be merged.

F1.3 Access requirements for reserved and unallocated registers

This section describes the access requirements for reserved and unallocated memory-mapped components.

The following information relates to certain types of reserved accesses:

- Reads and writes of unallocated locations. These accesses are reserved for the architecture.
- Reads and writes of locations for features that are not implemented, including:
 - OPTIONAL features that are not implemented.
 - Breakpoints and watchpoints that are not implemented.
 - Performance Monitors counters that are not implemented.
 - CTI triggers that are not implemented.
 - Error records that are not implemented.

These accesses are reserved.

- Reads of WO locations. These accesses are reserved for the architecture.
- Writes to RO locations. These accesses are reserved for the architecture.

Reserved accesses are normally RAZ/WI. However, software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture. Software must treat reserved accesses as RES0.

Chapter F2

System Level Implementation of the Generic Timer

This chapter defines the system level implementation of the Generic Timer. It contains the following sections:

- [About the Generic Timer specification](#)
- [Memory-mapped counter module](#)
- [Memory-mapped timer components](#)

Note

- [Generic Timer memory-mapped register descriptions](#) describes the System level Generic Timer registers. These registers are memory-mapped.
 - Additional Information for Implementations of the Generic Timer gives additional information, that does not form part of the architectural definition of a system level implementation of the Generic Timer.
 - [The Generic Timer in AArch32 state](#) gives a general description of the AArch32 state view of the Generic Timer, and describes the AArch32 System register interface to the Generic Timer.
-

F2.1 About the Generic Timer specification

[The Generic Timer in AArch32 state](#) describes the Arm Generic Timer and its implementation as seen from AArch32 state. These chapters include the definition of the low-latency System register interface to the Generic Timer. However, the Arm Generic Timer architecture also defines a memory-mapped component, that comprises:

- A memory-mapped counter module, that controls the generation of the Count value used by the Generic Timer. This memory-mapped counter module is required in any Arm Generic Timer implementation that requires software control of the Count value of the Generic Timer.
- Optional memory-mapped timer modules. These give a standardized way of providing timers for programmable system components other than PEs that implement the Arm architecture.

[The full set of Generic Timer components](#) summarizes them as seen from AArch32 state. [The system level components of the Generic Timer](#) summarizes the system level components.

F2.1.1 Registers in the system level implementation of the Generic Timer

Registers that control components of the system level implementation of the Generic Timer are grouped into *frames*. This specification defines the registers in each frame, and their offsets within the frame. The system defines the position of each frame in the memory map. This means the base addresses for each frame are IMPLEMENTATION DEFINED.

Note

The final 12 words of the first or only 4KB block of a register memory frame is an ID block.

Each frame must be in its own memory page, or memory protection region, and must be aligned to the size of the translation granule or protection granule.

Note

When a system level implementation of the Generic Timer is accessed by a PE:

- Using a PMSA, each frame is in its own memory protection region, aligned to the size of the memory protection granule.
-

The following sections give more information about the requirements for the system level Generic Timer component:

- [Supported access sizes](#).
- [Power and reset domains for the system level implementation of the Generic Timer](#).

F2.1.1.1 Supported access sizes

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the memory-mapped Generic Timer registers implementations must:

- Comply with the requirements of [Supported access sizes and Endianness](#).
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register, even if all components with direct memory access to the Generic Timer support making 64-bit accesses.

F2.1.1.2 Power and reset domains for the system level implementation of the Generic Timer

The power and reset domains of the system level implementation of the Generic Timer are IMPLEMENTATION DEFINED as part of the system implementation. In register descriptions, they are called Timer resets to indicate they can be outside the PE power and reset domains defined by the remainder of this manual.

The Arm architecture requires that the [CNTCR](#).{FCREQ, EN} and [CNTSR](#).FCACK fields reset to 0. These Timer reset values apply only on powerup of the power domain in which the registers are implemented or a reset of the reset domain in which they are implemented.

Every other register, or register field, of a system level implementation of the Generic Timer resets to a value that is architecturally UNKNOWN if it has a meaningful reset value. These Timer resets apply on powerup of the power domain in which the register is implemented, and on a reset of the reset domain in which it is implemented.

F2.1.2 The system level components of the Generic Timer

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, [CNTControlBase](#).
- A status frame, [CNTReadBase](#).

[Memory-mapped counter module](#) describes this component.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, [CNTCTLBase](#).

[The CNTCTLBase frame](#) describes this frame.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the [CNTBase \$N\$](#) frame.
- Optionally provides an unprivileged view of the timer in the [CNTEL0Base \$N\$](#) frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

[The CNTBase \$N\$ and CNTEL0Base \$N\$ frames](#) describes these frames.

F2.2 Memory-mapped counter module

The memory-mapped counter module provides top-level control of the system counter. The [CNTControlBase](#) frame holds the registers for the memory-mapped counter, and provides:

- An RW control register, [CNTCR](#), that provides:
 - An enable bit for the system counter.
 - An enable bit for Halt-on-debug. For more information, see [Halt-on-debug](#).
 - A field that can be written to request a change to the [Counter resolution](#). This mechanism means that, for example, if the [Counter resolution](#) is halved then the increment to the counter at each tick of the clock is doubled.
For more information, see [Control of counter resolution and increment](#).

Writes to this register are rare.

- A RO status register, [CNTSR](#), that provides:
 - A bit that indicates whether the system counter is halted because of an asserted Halt-on-debug signal.
 - A field that indicates whether a requested change to the [Counter resolution](#) has completed.
- Two contiguous 32-bit RW registers that hold the current system counter value, [CNTCV](#). If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.
The system counter must be disabled before writing to these registers, otherwise the effect of the write is UNPREDICTABLE.
Writes to these registers are rare.
- A *Frequency modes table* of one or more 32-bit entries, where:
 - The first entry in the table defines the *base frequency* of the system counter. It is IMPLEMENTATION DEFINED whether the base frequency indicates the effective frequency of the counter, or the maximum [Counter resolution](#).
Arm recommends that the base frequency indicates the maximum [Counter resolution](#).
 - Each subsequent entry in the table defines an alternative frequency of the system counter, that must be an exact divisor of the base frequency.

A 32-bit zero entry immediately follows the last table entry.

This table can be RO or RW. For more information, see [The Frequency modes table](#).

In an implementation with multiple physical address spaces, [CNTControlBase](#) is only accessible by accesses from the most secure physical address space. For example, in a system with Secure and Non-secure memory maps, [CNTControlBase](#) is only accessible by Secure accesses.

In addition, the [CNTReadBase](#) frame includes a read-only copy of the system counter value, [CNTCV](#), as two contiguous 32-bit RO registers. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

[Counter module control and status register summary](#) describes [CNTReadBase](#) and [CNTControlBase](#) memory maps, and the registers in each frame.

F2.2.1 Control of counter resolution and increment

The system counter has a fixed *base frequency*, and must maintain the required counter accuracy, meaning Arm recommends that it does not gain or lose more than ten seconds in a 24-hour period, see The system counter. However, the counter can operate at a lower resolution than the base frequency, using a correspondingly larger increment. For example, it can increment by a value four times larger when running at a quarter of the base frequency. Any lower resolution operation, and any switching between resolutions, must not reduce the accuracy of the counter.

Control of the system [Counter resolution](#) and increment is provided only through the memory-mapped counter module. The following sections describe this control:

- [The Frequency modes table.](#)
- [Changing the system counter and increment.](#)

F2.2.1.1 The Frequency modes table

The Frequency modes table starts at offset `0x20` in the `CNTControlBase` frame.

Table entries are 32-bits, and each entry specifies a selected frequency, in Hz.

The first entry in the table specifies the base frequency of the system counter.

When the system counter is operating at a lower frequency than the base frequency, the increment applied at each counter update is increased by the ratio of the base frequency and the selected frequency.

Note

Where the [Effective frequency](#) and the base frequency are not the same, the increment is not solely defined by the ratio of the base frequency and the selected frequency. For example, a counter with an [Effective frequency](#) of 1GHz and a base frequency of 125MHz will increment by a value of 8 on each tick of the base frequency. If the Frequency modes table provides an additional selected frequency of 31.25MHz, then when this frequency is selected the counter increments by 32 on each tick of the selected frequency.

A 32-bit word of zero value marks the end of the table. That is, the word of memory immediately after the last entry in the table must be zero.

The only required entry in the table is the entry for the base frequency.

Typically, the Frequency modes table is in RO memory. However, a system implementation might use RW memory for the table, and initialize the table entries as part of its startup sequence. Therefore, the `CNTControlBase` memory map shows the table region as RO or RW.

A system might provide an IMPLEMENTATION DEFINED mechanism for selecting the current power mode and resolution of the counter, and such a mechanism is independent of the memory-mapped counter module. If such a mechanism is provided, typically the frequency modes table only includes a single entry to indicate the base frequency.

Arm strongly recommends that the Frequency modes table is not updated once the system is running.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED, up to this limit.

Note

- Arm considers it likely that implementations will require significantly fewer entries than the architectural limit.
 - In the `CNTControlBase` frame, the offset range `0x0C0-0x0FC` can be used for IMPLEMENTATION DEFINED registers. If any registers are defined in this space, then the Frequency modes table cannot extend beyond offset `0x0B8`, with a zero word at offset `0x0BC`. This means that if any IMPLEMENTATION DEFINED registers are defined the maximum number of entries in the table is 40, including the zero-word end marker.
-

F2.2.1.2 Changing the system counter and increment

The value of the `CNTCR.FCREQ` field specifies which entry in the Frequency modes table specifies the selected frequency.

Changing the value of `CNTCR.FCREQ` requests a change to the selected frequency. To ensure the frequency change does not affect the overall accuracy of the counter, a change is made as follows:

- When changing from a higher frequency to a lower frequency, the counter:
 1. Continues running at the higher frequency until the count reaches an integer multiple of the required lower frequency.
 2. Switches to operating at the lower frequency.
- When changing from a lower frequency to a higher frequency, the counter:
 1. Waits until the end of the current lower-frequency cycle.
 2. Makes the counter increment required for operation at that lower frequency.
 3. Switches to operating at the higher frequency.

When the frequency has changed, [CNTSR](#) is updated to indicate the new selected frequency. Therefore, a system component that is waiting for a frequency change can poll [CNTSR](#) to detect the change.

F2.2.2 Halt-on-debug

The [CNTCR](#) register provides an enable bit for an OPTIONAL Halt-on-debug signal.

When the [CNTCR.HDBG](#) bit is set to 1, and the Halt-on-debug signal is implemented and asserted, the system counter is halted. Otherwise, the system counter ignores the state of this signal.

Where the system counter implements a Halt-on-debug signal and the system supports halting the system counter, Arm recommends that the Halt-on-debug signal can be controlled by a debugger using the Embedded Cross-Trigger (ECT) using a system-level cross-trigger interface that includes:

- A debug request output trigger event that asserts the Halt-on-debug signal.
- A restart request output trigger event that deasserts the Halt-on-debug signal.

For more information, see [About the Embedded Cross-Trigger](#).

Note

Software must use the Halt-on-debug enable bit to ensure that the timers cannot be halted maliciously in an attempt to prohibit progress.

For more information about Halt-on-debug, contact Arm.

F2.2.3 Counter module control and status register summary

The Counter module control and status registers are memory-mapped registers in the following register memory *frames*:

- A control frame, with base address [CNTControlBase](#).
- A status frame, with base address [CNTReadBase](#).

Each of these register memory frames is in its own memory page or [memory protection region](#), and the frame base address points to the start of this region. Each base address must be aligned to the size of the translation granule or protection granule.

Note

Each frame of a memory-mapped Generic Timer takes the name of its base address.

In each register memory frame, the memory at offset `0xFD0-0xFFF` is reserved for twelve 32-bit IMPLEMENTATION DEFINED ID registers, see the [CounterID<n>](#) register descriptions for more information.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the counter is little-endian.

In an implementation with multiple physical address spaces, [CNTControlBase](#) is only accessible by accesses from the most secure physical address space. For example, in a system with Secure and Non-secure memory maps, [CNTControlBase](#) is only accessible by Secure accesses.

Table F2-1 shows the [CNTControlBase](#) control registers, in order of their offsets from the [CNTControlBase](#) base address, for an implementation that includes registers in the IMPLEMENTATION DEFINED register space 0x0C0-0x0FC, and also has fewer than 39 [CNTFID<n>](#) registers. The [Frequency modes table](#) describes how this memory map differs if more [CNTFID<n>](#) registers are implemented.

[Generic Timer memory-mapped register descriptions](#) describes each of these registers.

Table F2-1 CNTControlBase memory map

Offset	Name	Type	Description
0x000	CNTCR	RW	Counter Control Register.
0x004	CNTSR	RO	Counter Status Register.
0x008	CNTCV[31:0]	RW	Counter Count Value register.
0x00C	CNTCV[63:32]	RW	
0x010	CNTSCR^a	RW	Counter Scale Register.
0x014-0x018	-	RES0	Reserved.
0x01C	CNTID^a	RO	Counter Identification Register.
0x020	CNTFID0	RO or RW	Frequency modes table, and end marker. For more information, see The Frequency modes table .
0x020+4 <i>n</i>	CNTFID<n>	RO or RW	
0x024+4 <i>n</i>	-	RO or RW, RAZ	
(0x028+4 <i>n</i>)- 0x0BC	-	RO, RES0	Reserved.
0x0C0-0x0FC	-	IMPLEMENTATION DEFINED	Reserved for IMPLEMENTATION DEFINED registers.
0x100-0xFCC	-	RO, RES0	Reserved.
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11.

a. Implemented only if FEAT_CNTSC is implemented.

Table F2-2 shows the [CNTReadBase](#) control registers, in order of their offsets from the [CNTReadBase](#) base address. [Generic Timer memory-mapped register descriptions](#) describes each of these registers.

Table F2-2 CNTReadBase memory map

Offset	Name	Type	Description
0x000	CNTCV[31:0]	RO	Counter Count Value register
0x004	CNTCV[63:32]	RO	
0x008-0xFCC	-	RES0	Reserved
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11

F2.3 Memory-mapped timer components

This part of the Arm Generic Timer specification defines an optional memory-mapped timer component. This can be implemented as part of any programmable system component that does not incorporate a System register mapped Arm Generic Timer, to provide that system component with the timer functionality of an Arm Generic Timer.

The memory map consists of up to eight timer *frames*. The base address of a frame is `CNTBaseN`, where *N* numbers from 0 up to a maximum permitted value of 7.

Each `CNTBaseN` timer frame:

- Provides its own set of timers and associated interrupts.
- Is implemented in its own memory page or [memory protection region](#).
- Is implemented at a base address, identified as `CNTBaseN`, that is aligned to the size of the translation granule or [memory protection region](#).

For each implemented `CNTBaseN` frame the system can optionally provide an unprivileged view of the frame, described as the EL0 view of the frame. The base address of this second view of the `CNTBaseN` frame is `CNTELOBaseN`.

Note

In the naming of the registers associated with a `CNTBaseN` or `CNTELOBaseN` frame, the value of *N* is represented as `<n>`, for example `CNTACR<n>`.

If a `CNTELOBaseN` frame is implemented:

- Is implemented in its own memory page or [memory protection region](#) and is aligned to the size of the translation granule or [memory protection region](#).
- All registers visible in `CNTBaseN`, except for `CNTVOFF` and `CNTELOACR`, can be visible in `CNTELOBaseN`.
 - Control fields in `CNTELOACR` determine whether each register is visible.
- The offsets of all visible registers are the same as their offsets in the `CNTBaseN` frame.

In addition to the implemented `CNTBaseN` and `CNTELOBaseN` frames, the system must provide a single control frame at base address `CNTCTLBase`. `CNTCTLBase` must be implemented in its own memory page or [memory protection region](#) and is aligned to the size of the translation granule or [memory protection region](#).

The system defines the position of each frame in the memory map. This means the values of each of the `CNTBaseN`, `CNTELOBaseN`, and `CNTCTLBase` base addresses is IMPLEMENTATION DEFINED.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the memory-mapped timers are little-endian.

The following sections describe the implementation of a memory-mapped view of the counter and timer:

- [The CNTCTLBase frame.](#)
- [The CNTBaseN and CNTELOBaseN frames.](#)

Note

Providing a complete set of features in a system level implementation gives an implementation example for a system level Generic Timer implementation that provides equivalent features to a System registers Generic Timer implementation in a PE that includes all of the Exception levels.

F2.3.1 The CNTCTLBase frame

The `CNTCTLBase` frame contains:

- An identification register for the features of the memory-mapped counter and timer implementation.

- Access controls for each [CNTBaseN](#) frame.
- A virtual offset register for frames that implement a virtual timer.

[Table F2-3](#) shows the [CNTCTLBase](#) registers, in order of their offsets from the [CNTCTLBase](#) base address.

Note

[CNTFRQ](#) and [CNTVOFF](#) registers are also implemented in a System register interface to the Generic Timer.

[Generic Timer memory-mapped register descriptions](#) describes each of these registers.

Table F2-3 CNTCTLBase memory map

Offset	Register	Type	Security ^a	Description
0x000	CNTFRQ^b	RW	Restricted	Counter Frequency register.
0x004	CNTNSAR	RW	Restricted	Counter Non-Secure Access register.
0x008	CNTTIDR	RO	No restrictions	Counter Timer ID register.
0x00C- 0x03F	-	RES0	-	Reserved.
0x040+4N ^c	CNTACR<n>	RW	Programmable ^d	Counter Access Control register <i>N</i> .
0x060- 0x07F	-	RES0	-	Reserved.
0x080+8N ^c	CNTVOFF<n>[31:0]^b	RW ^e	Programmable ^d	Virtual Offset register <i>N</i> . If the CNTBaseN frame has virtual timer capability then CNTVOFF<n> is implemented as an RW register, otherwise its location is RAZ/WI.
0x084+8N ^c	CNTVOFF<n>[63:32]^b	RW ^e		
0x0C0-0x0FC	-	RES0	-	Reserved.
0x100-0x7FC	-	-	-	IMPLEMENTATION DEFINED.
0x800-0xFBC	-	RES0	-	Reserved.
0xFC0-0xFCF	-	-	-	IMPLEMENTATION DEFINED.
0xFD0- 0xFFC	CounterID<n>	RO	No restrictions	Counter ID registers 0-11.

a. Access security restrictions in an implementation with multiple Security states. See the individual register descriptions and [CNTNSAR](#) for details on the restrictions. In an implementation that supports only a single Security state all registers are accessible as shown in the *Type* column.

b. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in [Generic Timer registers](#) and [Generic Timer registers](#). The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

c. Implemented for each value of *N* from 0 to 7 for which a [CNTBaseN](#) frame is implemented.

d. The [CNTNSAR](#) determines the Non-secure accessibility of the [CNTACR<n>](#) and the [CNTVOFF<n>](#) in the [CNTCTLBase](#) frame. For more information, see the register descriptions.

e. Address is reserved, RAZ/WI if register not implemented.

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the [CNTVOFF](#) register that defines the virtual offset between the physical count and the virtual count. If a memory-mapped Generic Timer component does not distinguish between real time and virtual time, then it can implement [CNTVOFF](#) as RAZ/WI. Otherwise [CNTVOFF](#) is an RW register, and Arm strongly recommends that the system only permits access to [CNTVOFF](#) from EL2 or higher.

F2.3.2 The CNTBaseN and CNTEL0BaseN frames

Each [CNTBaseN](#) frame, or {[CNTBaseN](#), [CNTEL0BaseN](#)} pair of frames, provides a memory-mapped counter and timer, see:

- The CNTBaseN frame.
- The CNTEL0BaseN frame.
- CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames.

F2.3.2.1 The CNTBaseN frame

Table F2-4 shows the CNTBaseN registers, in order of their offsets from the CNTBaseN base address. Whether a frame includes a virtual timer is IMPLEMENTATION DEFINED. If it does not, then memory at offsets 0x030-0x03C is RAZ/WI. Except for CNTEL0ACR and the CounterID<n> registers, equivalent registers are also implemented in a System register interface to the timer component of a Generic Timer.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table F2-4 CNTBaseN memory map

Offset	Register	Type	Description
0x000	CNTPCT[31:0] ^a	RO	Physical Count register.
0x004	CNTPCT[63:32] ^a	RO	
0x008	CNTVCT[31:0] ^a	RO	Virtual Count register.
0x00C	CNTVCT[63:32] ^a	RO	
0x010	CNTRFQ ^a	RO ^c	Counter Frequency register.
0x014	CNTEL0ACR	RW ^b	Counter EL0 Access Control Register, optional in the CNTBaseN memory map.
0x018	CNTVOFF[31:0] ^a	RO ^c	Virtual Offset register. If CNTVOFF in the CNTCTLBase frame is an RW register, a read of this register returns the value of that register. Otherwise is RAZ.
0x01C	CNTVOFF[63:32] ^a	RO ^c	
0x020	CNTP_CVAL[31:0] ^a	RW	Physical Timer CompareValue register.
0x024	CNTP_CVAL[63:32] ^a	RW	
0x028	CNTP_TVAL ^a	RW	Physical TimerValue register.
0x02C	CNTP_CTL ^a	RW	Physical Timer Control register.
0x030	CNTV_CVAL[31:0] ^a	RW ^b	Virtual Timer CompareValue register, optional in the CNTBaseN memory map.
0x034	CNTV_CVAL[63:32] ^a	RW ^b	
0x038	CNTV_TVAL ^a	RW ^b	Virtual TimerValue register, optional in the CNTBaseN memory map.
0x03C	CNTV_CTL ^a	RW ^b	Virtual Timer Control register, optional in the CNTBaseN memory map.
0x040-0xFFC	-	RES0	Reserved.
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11.

a. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in Generic Timer registers and Generic Timer registers. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

b. Address is reserved, RAZ/WI if register not implemented.

c. The CNTCTLBase frame includes an RW view of this register.

F2.3.2.2 The CNTEL0BaseN frame

For any value of N, the layout of the registers in the CNTEL0BaseN frame is identical to the CNTBaseN frame, except that, in the CNTEL0BaseN frame:

- CNTVOFF is never visible, and the memory at 0x018-0x01C is RAZ/WI.

- **CNTELOACR** is never visible, and the memory at 0x014 is RAZ/WI.
- If implemented in the **CNTBaseN** frame, **CNTELOACR** controls whether **CNTPCT**, **CNTVCT**, **CNTRFQ**, the Physical Timer, and the Virtual Timer registers are visible in the **CNTELOBaseN** frame.
If **CNTELOACR** is not implemented then these registers are not visible in the **CNTELOBaseN** frame, and their addresses in that frame are RAZ/WI.

If an implementation supports 64-bit atomic accesses, then **CNTPCT**, **CNTVCT**, **CNTVOFF**, **CNTP_CVAL**, and **CNTV_CVAL** must be accessible as atomic 64-bit values.

F2.3.2.3 CNTCTLBase status and control fields for the CNTBaseN and CNTELOBaseN frames

In the **CNTCTLBase** frame:

CNTTIDR controls:

- Whether each **CNTBaseN** frame is implemented.
- If a **CNTBaseN** frame is implemented, whether:
 - That **CNTBaseN** frame has virtual timer capability.
 - A corresponding **CNTELOBaseN** frame is implemented.

CNTNSAR controls:

Whether all the following are accessible by Non-secure accesses:

- **CNTACR<n>**.
- **CNTVOFF<n>**.
- Each implemented **CNTBaseN** frame, and any corresponding **CNTELOBaseN** frame.

The **CNTACR<n>** registers control:

For each implemented **CNTBaseN** frame, the accessibility of the following registers in that frame:

- **CNTP_CTL**, **CNTP_CVAL**, and **CNTP_TVAL**.
- **CNTV_CTL**, **CNTV_CVAL**, and **CNTV_TVAL**.
- **CNTVOFF**.
- **CNTRFQ**.
- **CNTPCT**.
- **CNTVCT**.

For **CNTACR<n>**, the value of <n> corresponds to the value of *N* for the controlled **CNTBaseN** frame.

The **CNTVOFF<n>** registers provide:

For each implemented **CNTBaseN** frame that has virtual capability, the RW copy of the **CNTVOFF** register for that frame.

Note

In a **CNTBaseN** frame that has virtual timer capability the **CNTVOFF** register is RO.

For **CNTVOFF<n>**, the value of <n> corresponds to the value of *N* for the controlled **CNTBaseN** frame.

Chapter F3

Recommended External Interface to the Performance Monitors

This chapter describes the recommended external interface to the Performance Monitors. It contains the following section:

- [About the external interface to the Performance Monitors registers](#)

Note

[Performance Monitors external register descriptions](#) describes the external view of the Performance Monitors registers.

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

Arm recommends that:

- An implementation provides the OPTIONAL external debug interface to the Performance Monitors registers, [FEAT_PMUv3_EXT](#).

Note

A debugger can use this interface to access counters in the Performance Monitors.

- The implementation includes the OPTIONAL support for memory-mapped access to the External debug interface.

Note

— Software running on any PE in a system can use this interface to access counters in the Performance Monitors.

— Privileged software should use the MPU to control access to this interface.

- The external debug interface is implemented as defined in Recommended External Debug Interface.

The following sections describe the memory-mapped views of the Performance Monitors registers:

- [Differences in the external views of the Performance Monitors registers](#).
- [Access permissions for external views of the Performance Monitors](#).

In this section, unless the context explicitly indicates otherwise, any reference to *a memory-mapped view* applies equally to a register view using:

- An access through an external debug interface.
- A memory-mapped access.

F3.1.1 Supported access sizes

[External Performance Monitors registers summary](#) gives the memory map of these registers.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the external interface to the Performance Monitors registers implementations must:

- Comply with the requirements of [Supported access sizes and Endianness](#).
- When [FEAT_PMUv3_EXT32](#) is implemented, support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register, even if all components with direct memory access to the PMU support making 64-bit accesses.

Permitted word-aligned 32-bit accesses are single-copy atomic at word granularity.

F3.1.2 Differences in the external views of the Performance Monitors registers

An external view of the Performance Monitors registers accesses the same registers as the System registers interface described in [Performance Monitors Extension registers](#), except that:

- The PMSELR is accessible only in the System registers interface.
- The following registers are accessible only in external views:
 - [PMAUTHSTATUS](#).
 - [PMCCR](#).
 - [PMCFGR](#).
 - [PMCIDR0](#).
 - [PMCIDR1](#).
 - [PMCIDR2](#).

- [PMCIDR3](#).
- [PMDEVARCH](#).
- [PMDEVTYPE](#).
- [PMEVFILT2R<n>](#).
- [PMIIDR](#).
- [PMLAR](#).
- [PMLSR](#).
- [PMPCCTL](#).
- [PMPCSR](#).
- [PMPIDR0](#).
- [PMPIDR1](#).
- [PMPIDR2](#).
- [PMPIDR3](#).
- [PMPIDR4](#).
- [PMPIDR5](#).
- [PMPIDR6](#).
- [PMPIDR7](#).

[Performance Monitors external register descriptions](#) describes these registers.

- The following registers are accessible only in the external view when [FEAT_PMUv3_EXT32](#) is implemented:
 - [PMCID1SR](#).
 - [PMCID2SR](#).
 - [PMDEVAFF0](#).
 - [PMDEVAFF1](#).
 - [PMVIDSR](#).

[Performance Monitors external register descriptions](#) describes these registers.

- The following controls do not affect the external views:
 - [PMSELR](#).
 - [PMUSERENR](#).
 - [HDCR](#). {TPM, TPMCR, HPMN}.

Instead, see the register descriptions in [External System Control Register Descriptions](#).

- The [PMSWINC_ELO](#) register is OPTIONAL in the external view when [FEAT_PMUv3_EXT32](#) is implemented.

F3.1.3 Access permissions for external views of the Performance Monitors

IDZGKJ For more information, see [External debug interface register access permissions](#).

IYMBZZ The following table shows the access permissions for the Performance Monitors registers in a Debug implementation for Armv8-R implementations.

The table uses the following terms:

DLK	When FEAT_DoubleLock is implemented and locked, DoubleLockStatus() == TRUE, accesses to some registers produce an error. Applies to both interfaces.
EPMAD	When AllowExternalPMUAccess() == FALSE, external debug access is disabled for the access. See also Behavior of a not permitted memory-mapped access .
Error	Indicates that the access gives an error response.
Def	This shows the default access permissions, if none of the conditions in this list prevent access to the register.
Off	The Core power domain is completely off, or in a low-power state where the Core power domain registers cannot be accessed, and EDPRSR.PU will read as zero.

Note
If debug power is off, then all external debug interface accesses return an error.

OSLK	When the OS Lock is locked, OSLSR_EL1.OSLK is 1, accesses to some registers produces an error. This column shows the effect of this control on accesses using the external debug interface.
SLK	This indicates the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the optional Software Lock is locked. See Register access permissions for memory-mapped accesses . For all other accesses, this column is ignored.
-	Indicates that the control has no effect on the behavior of the access: - If no other control affects the behavior, the Default access behavior applies. - However, another control might determine the behavior.

Table F3-1 Access permissions for the Performance Monitors registers when FEAT_PMuV3_EXT32 is implemented

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	Def	SLK
0x000+8xn 0x004+8xn	PMEVCNTR<n>_EL0[31:0] ^a PMEVCNTR<n>_EL0[63:32] ^a	Core	Error	Error	Error	Error	RW	RO
0x0F8 0x0FC	PMCCNTR_EL0[31:0] PMCCNTR_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO
0x100 0x104	PMICNTR_EL0[31:0] PMICNTR_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO
0x200 0x204	PMPCSR[31:0] PMPCSR[63:32]	Core	Error	Error	Error	-	RO	RO
0x208	PMCID1SR	Core	Error	Error	Error	-	RO	RO
0x20C	PMVIDSR	Core	Error	Error	Error	-	RO	RO
0x220 0x224	PMPCSR[31:0] PMPCSR[63:32]	Core	Error	Error	Error	-	RO	RO
0x228	PMCID1SR	Core	Error	Error	Error	-	RO	RO
0x22C	PMCID2SR	Core	Error	Error	Error	-	RO	RO
0x400+4xn	PMEVTYPER<n>_EL0 ^a [31:0]	Core	Error	Error	Error	Error	RW	RO
0x47C	PMCCFILTR_EL0[31:0]	Core	Error	Error	Error	Error	RW	RO
0x480	PMICFILTR_EL0[31:0]	Core	Error	Error	Error	Error	RW	RO
0x600 - 0x7FC	IMPLEMENTATION DEFINED							
0x800+4xn	PMEVFILT2R<n>	Core	Error	Error	Error	Error	RW	RO
0xA00+4xn	PMEVTYPER<n>_EL0 [63:32]	Core	Error	Error	Error	Error	RW	RO
0xA7C	PMCCFILTR_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO
0xA80	PMICFILTR_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO
0xA84- 0xAFC	Reserved	-						
0xB00- 0xBFC	-	-	IMPLEMENTATION DEFINED registers ^d					
0xC00 0xC04	PMCNTENSET_EL0[31:0] PMCNTENSET_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO
0xC20 0xC24	PMCNTENCLR_EL0[31:0] PMCNTENCLR_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	Def	SLK
0xC40 0xC44	PMINTENSET_EL1[31:0] PMINTENSET_EL1[63:32]	Core	Error	Error	Error	Error	RW	RO
0xC60 0xC64	PMINTENCLR_EL1[31:0] PMINTENCLR_EL1[63:32]	Core	Error	Error	Error	Error	RW	RO
0xC80 0xC84	PMOVSCLR_EL0[31:0] PMOVSCLR_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO
0xCA0	PMSWINC_EL0	Core	Error	Error	Error	Error	WO	WI
0xCA0 0xCA4	PMZR_EL0[31:0] PMZR_EL0[63:32]	Core	Error	Error	Error	Error	WO	WI
0xCC0 0xCC4	PMOVSSET_EL0[31:0] PMOVSSET_EL0[63:32]	Core	Error	Error	Error	Error	RW	RO
0xCE0	PMCGCR0	Core	Error	Error	Error	Error	RO	RO
0xD80- 0xDFC	-	-	IMPLEMENTATION DEFINED registers ^d					
0xE00	PMCFGR	Core	Error	Error	Error	Error	RO	RO
0xE04	PMCR_EL0	Core	Error	Error	Error	Error	RW	RO
0xE08	PMIIDR	Core	Error	Error	Error	Error	RO	RO
0xE20 0xE24 0xE28 0xE2C	PMCEID0 PMCEID1 PMCEID2 PMCEID3	Core	Error	Error	Error	Error	RO	RO
0xE30- 0xE34	IMPLEMENTATION DEFINED							
0xE38- 0xE3C		-	IMPLEMENTATION DEFINED registers ^b					
0xE40 0xE44	PMMIR[31:0] PMMIR[63:32]	Core	Error	Error	Error	Error	RO	RO
0xE50 0xE54	PMPCSTL[31:0] PMPCSTL[63:32]	Core	Error	Error	Error	-	RW	RW
0xE58 ^c	PMCCR	Core	Error	Error	-	-	RW	RW
0xE80- 0xEFC	Integration registers	-	IMPLEMENTATION DEFINED registers ^b					
0xF00- 0xFFC	Management registers and CoreSight compliance							

a. Implemented event counters only. *n* is the counter number.

b. See IMPLEMENTATION DEFINED registers.

c. The default access permissions for this location are RAZ/WI.

Chapter F4

External System Control Register Descriptions

This chapter describes the external system control registers. It excludes the External debug registers that are described in [External Debug Register Descriptions](#). It contains the following sections:

- [About the external system control register descriptions](#)
- [External Performance Monitors registers summary](#)
- [Performance Monitors external register descriptions](#)
- [Generic Timer memory-mapped registers overview](#)
- [Generic Timer memory-mapped register descriptions](#)

F4.1 About the external system control register descriptions

This chapter describes the external system control registers other than the external debug registers. That is, it describes:

An external view of the Performance Monitors registers

Arm recommends that implementations provide access to the Performance Monitors registers through the OPTIONAL External debug interface, and provide the OPTIONAL memory-mapped interface to this interface:

- [External Performance Monitors registers summary](#) lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
- [Performance Monitors external register descriptions](#) describes each of the memory-mapped registers.

[Recommended External Interface to the Performance Monitors](#) describes the recommended interface to these registers.

Note

[The Performance Monitors Extension](#) describes the Performance Monitors. The following section describes the System register interfaces to the Performance Monitors, [Performance Monitors registers](#), for accesses from an Exception level that is using AArch32.

The registers for the system level Generic Timer component

Any implementation that includes the Generic Timer must include the memory-mapped system level component described in [System Level Implementation of the Generic Timer](#). In this chapter:

- [Generic Timer memory-mapped registers overview](#) gives an overview of the registers, referring to [System Level Implementation of the Generic Timer](#) for more information.
- [Generic Timer memory-mapped register descriptions](#) describes each of the memory-mapped registers.

Note

[The Generic Timer in AArch32 state](#) describes the Generic Timer component that is accessible using the System registers. The following section describes the System register interfaces to that component, [Generic Timer registers](#), for accesses from an Exception level that is using AArch32.

Note

[External Debug Register Descriptions](#) describes the external debug registers.

F4.2 External Performance Monitors registers summary

When an implementation provides access to the Performance Monitors registers through the External debug interface, that interface provides access to:

- Performance Monitors registers.
- A read-only configuration register, [PMCFGR](#).
- The OPTIONAL CoreSight registers for the Performance Monitors, if they are implemented.

The locations of the registers are defined as offsets from a system-defined base address. [Performance Monitors external register views](#) defines this memory map.

F4.2.1 Performance Monitors external register views

The following tables show the external view of the Performance Monitors registers:

- [Table F4-1](#) when [FEAT_PMUv3_EXT32](#) is implemented.

All other entries are reserved.

Note	
•	Counters that are reserved because HDCR.HPMN has been changed from its reset value remain visible in any external view.
•	The registers that relate to an implemented event counter, PMNx, are PMEVCNTR<n> and PMEVTYPER<n>.
•	Tables in this section only list the Armv8 registers. For encoding information of the registers introduced by Armv9, see the individual register descriptions in this chapter.

Each entry in the Name column links to the register description in [Performance Monitors external register descriptions](#).

Table F4-1 Performance Monitors external register views when FEAT_PMUv3_EXT32 is implemented

Name	Type	Description	Offset
PMEVCNTR<n>_ELO[31:0] PMEVCNTR<n>_ELO[63:32]	RW	Performance Monitors Event Counter Register	0x000+8xn 0x004+8xn
PMCCNTR_ELO[31:0] PMCCNTR_ELO[63:32]	RW	Performance Monitors Cycle Counter Register ^a	0x0F8 0x0FC
PMEVTYPER<n>_ELO [31:0]	RW	Performance Monitors Event Type and Filter Register	0x400+4xn
PMCCFILTR_ELO [31:0]	RW	Performance Monitors Cycle Counter Filter Register	0x47C
PMEVFILT2R<n>	RW	Performance Monitors Event Filter Registers	0x800+8xn
PMEVTYPER<n>_ELO [63:32]	RW	Performance Monitors Event Type and Filter Register	0xA00+4xn
PMCCFILTR_ELO [63:32]	RW	Performance Monitors Cycle Counter Filter Register	0xA7C
-	-	IMPLEMENTATION DEFINED	0xB00-0xBF C
PMCNTENSET_ELO[31:0] PMCNTENSET_ELO[63:32]	RW	Performance Monitors Count Enable Set Register	0xC00 0xC04
PMCNTENCLR_ELO[31:0] PMCNTENCLR_ELO[63:32]	RW	Performance Monitors Count Enable Clear Register	0xC20 0xC24
PMINTENSET_EL1[31:0] PMINTENSET_EL1[63:32]	RW	Performance Monitors Interrupt Enable Set Register	0xC40 0xC44

Name	Type	Description	Offset
PMINTENCLR_EL1[31:0] PMINTENCLR_EL1[63:32]	RW	Performance Monitors Interrupt Enable Clear Register	0xC60 0xC64
PMOVSCLR_EL0[31:0] PMOVSCLR_EL0[63:32]	RW	Performance Monitors Overflow Flag Status Clear Register	0xC80 0xC84
PMSWINC_EL0 ^b	WO	Performance Monitors Software Increment Register	0xCA0
PMOVSSET_EL0[31:0] PMOVSSET_EL0[63:32]	RW	Performance Monitors Overflow Flag Status Set Register	0xCC0 0xCC4
-	-	IMPLEMENTATION DEFINED	0xD80-0xDFC
PMCFGR	RO	Performance Monitors Configuration Register	0xE00
PMCR_EL0	RW	Performance Monitors Control Register	0xE04
PMIIDR	RO	Performance Monitors Implementation Identification Register	0xE08
PMCEID0 PMCEID1	RO	Performance Monitors Common Event Identification Registers	0xE20 0xE24
-	-	IMPLEMENTATION DEFINED	0xE38-0xE3C
-	-	IMPLEMENTATION DEFINED	0xE80-0xEFC
PMITCTRL ^c	RW	Integration Model Control registers	0xF00
PMDEVAFF0 ^c PMDEVAFF1 ^c	RO	Device Affinity registers	0xFA8 0xFAC
PMLAR ^{c, d}	WO	Lock Access Register	0xFB0
PMLSR ^{c, d}	RO	Lock Status Register	0xFB4
PMAUTHSTATUS ^c	RO	Authentication Status Register	0xFB8
PMDEVARCH ^c	RO	Device Architecture Register	0xFBC
PMDEVTYPE ^c	RO	Device Type Register	0xFCC
PMPIDR4 ^c PMPIDR5 ^c PMPIDR6 ^c PMPIDR7 ^c PMPIDR0 ^c PMPIDR1 ^c PMPIDR2 ^c PMPIDR3 ^c	RO	Peripheral ID registers	0xFD0 0xFD4 0xFD8 0xFDC 0xFE0 0xFE4 0xFE8 0xFEC
PMCIDR0 ^c PMCIDR1 ^c PMCIDR2 ^c PMCIDR3 ^c	RO	Component ID registers	0xFF0 0xFF4 0xFF8 0xFFC

a. The interface must support at least single-copy atomic 32-bit accesses. If single-copy atomic 64-bit access to the registers is not possible, software must use a high-low-high read access to read the counter value if the counter is enabled.

b. OPTIONAL.

c. CoreSight interface registers.

d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

F4.3 Performance Monitors external register descriptions

This section describes the external view of the Performance Monitors registers. [External Performance Monitors registers summary](#) lists these registers in offset order.

F4.3.1 PMAUTHSTATUS, Performance Monitors Authentication Status register

The PMAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitors.

Configuration

This register is in the Debug power domain.

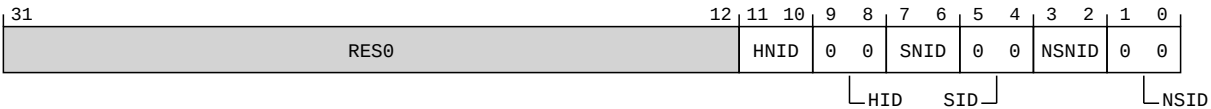
This register is OPTIONAL, and is required for CoreSight compliance. Arm recommends that this register is implemented.

This register is present only when FEAT_PMuV3_EXT is implemented and FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMAUTHSTATUS are RES0.

Attributes

PMAUTHSTATUS is a 32-bit register.

Field descriptions



Bits [31:12]

Reserved, RES0.

HNID, bits [11:10]

Holds the same value as [DBGAUTHSTATUS_EL1.HNID](#).

HID, bits [9:8]

Hyp invasive debug.

HID	Meaning
0b00	Not implemented.

All other values are reserved.

Access to this field is RO.

SNID, bits [7:6]

Holds the same value as [DBGAUTHSTATUS_EL1.SNID](#).

SID, bits [5:4]

Secure invasive debug.

SID	Meaning
0b00	Not implemented.

All other values are reserved.

Access to this field is RO.

NSNID, bits [3:2]

Holds the same value as [DBGAUTHSTATUS_EL1.NSNID](#).

NSID, bits [1:0]

Non-secure invasive debug.

NSID	Meaning
0b00	Not implemented.

All other values are reserved.

Access to this field is RO.

Accessing PMAUTHSTATUS

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0x`FB8` from PMU

Accesses to this register are RO.

F4.3.2 PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configuration

PMCCFILTR_EL0 is in the Core power domain.

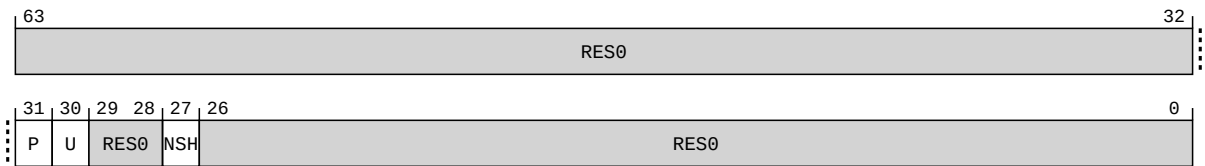
External register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to AArch32 System register PMCCFILTR[31:0].

This register is present only when FEAT_PMuV3_EXT is implemented and FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are RES0.

Attributes

PMCCFILTR_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting cycles in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL1.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL1 is further controlled by PMCCFILTR_EL0.NSK.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

U, bit [30]

EL0 filtering. Controls counting cycles in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL0.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL0 is further controlled by PMCCFILTR_EL0.NSU.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 filtering. Controls counting cycles in EL2.

NSH	Meaning
0b0	The PE does not count cycles in EL2.
0b1	This mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [26:0]

Reserved, RES0.

Accessing PMCCFILTR_EL0

Accesses to this register use the following encodings:

When FEAT_PMu3_EXT32 is implemented

[31:0] Accessible at offset 0x47C from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.3 PMCCNTR_EL0, Performance Monitors Cycle Counter

The PMCCNTR_EL0 characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. For more information, see ‘Time as measured by the Performance Monitors cycle counter’.

[PMCCFILTR_EL0](#) determines the modes and states in which the PMCCNTR_EL0 can increment.

Configuration

PMCCNTR_EL0 is in the Core power domain.

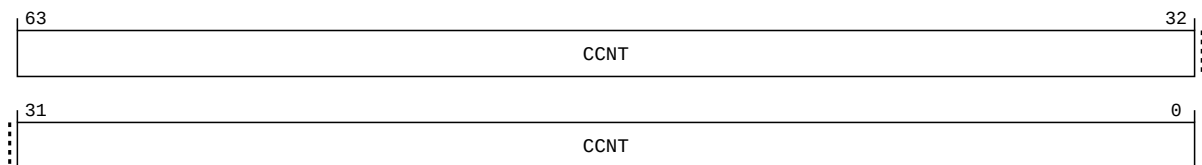
External register PMCCNTR_EL0 bits [63:0] are architecturally mapped to AArch32 System register [PMCCNTR](#)[63:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR_EL0 are RES0.

Attributes

PMCCNTR_EL0 is a 64-bit register.

Field descriptions



CCNT, bits [63:0]

Cycle count. Depending on the values of [PMCR_EL0](#).{LC,D}, the cycle count increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to [PMCR_EL0](#).C sets this field to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[63:0] Accessible at offset 0x0F8 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.4 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x0000 to 0x001F.

For more information about the Common events and the use of the PMCEIDn registers, see ‘[The PMU event number space and common events](#)’.

Use of this register is deprecated.

Note

This view of the register was previously called PMCEID0_EL0.

Configuration

PMCEID0 is in the Core power domain.

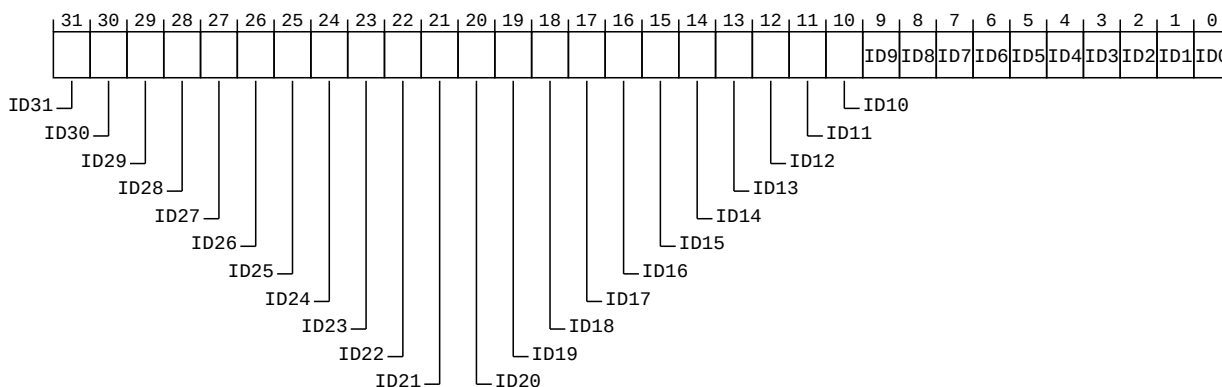
When FEAT_PMUv3_EXT32 is implemented, External register PMCEID0 bits [31:0] are architecturally mapped to AArch32 System register [PMCEID0](#)[31:0].

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID0 are RES0.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions



ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

ID<n>	Meaning
0b0	The Common event is not implemented, or not counted.

0b1

The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID0

Note

`AllowExternalPMUAccess()` has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xE20 from PMU

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `!AllowExternalPMUAccess(addrdesc)`, accesses to this register generate an error response.
- When `OSLockStatus()`, accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

F4.3.5 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0×020 to $0 \times 03F$.

For more information about the Common events and the use of the PMCEID n registers, see ‘[The PMU event number space and common events](#)’.

Use of this register is deprecated.

Note

This view of the register was previously called PMCEID1_EL0.

Configuration

PMCEID1 is in the Core power domain.

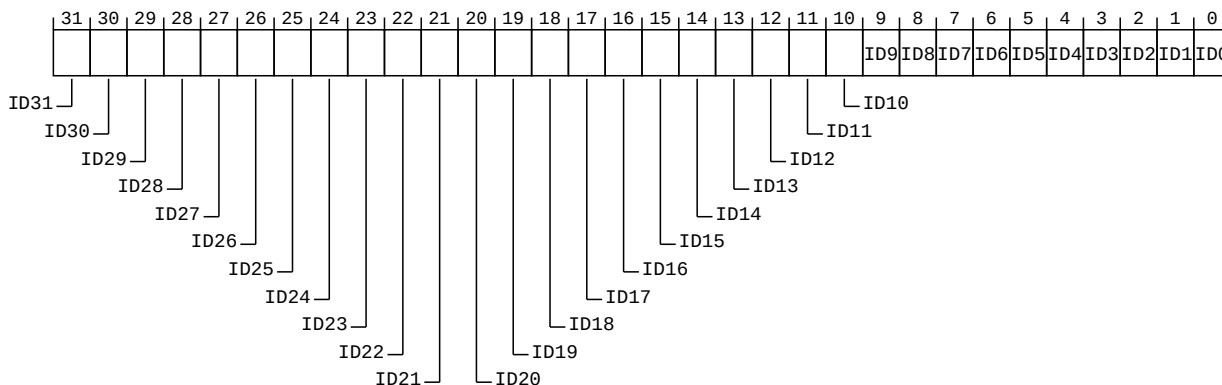
When FEAT_PMUv3_EXT32 is implemented, External register PMCEID1 bits [31:0] are architecturally mapped to AArch32 System register [PMCEID1](#)[31:0].

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID1 are RES0.

Attributes

PMCEID1 is a 32-bit register.

Field descriptions



ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event ($0 \times 0020 + n$).

For each bit:

ID<n>	Meaning
0b0	The Common event is not implemented, or not counted.

0b1

The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID1

Note

`AllowExternalPMUAccess()` has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xE24 from PMU

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `!AllowExternalPMUAccess(addrdesc)`, accesses to this register generate an error response.
- When `OSLockStatus()`, accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

F4.3.6 PMCFGR, Performance Monitors Configuration Register

The PMCFGR characteristics are:

Purpose

Contains PMU-specific configuration data.

Configuration

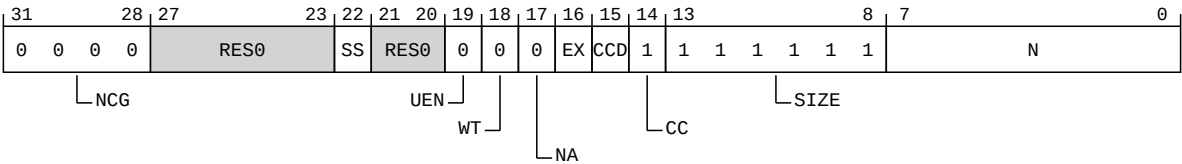
PMCFGR is in the Core power domain.

This register is present only when FEAT_PMuV3_EXT is implemented and FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMCFGR are RES0.

Attributes

PMCFGR is a 32-bit register.

Field descriptions



NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

NCG	Meaning
0b0000	One counter group implemented.

All other values are reserved.

Access to this field is RO.

Bits [27:23]

Reserved, RES0.

SS, bit [22]

Snapshot supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SS	Meaning
0b0	Snapshot mechanism not supported. The locations 0x600-0x7FC and 0xE30-0xE3C are IMPLEMENTATION DEFINED.

SS	Meaning
0b1	Snapshot mechanism supported. Locations 0x600-0x7FC and 0xE30-0xE3C contain IMPLEMENTATION DEFINED snapshot registers.

A PMU might include an IMPLEMENTATION DEFINED snapshot mechanism, including one using the IMPLEMENTATION DEFINED registers 0x600-0x7FC and 0xE30-0xE3C.

Access to this field is RO.

Bits [21:20]

Reserved, RES0.

UEN, bit [19]

User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

WT, bit [18]

This feature is not supported, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

NA, bit [17]

This feature is not supported, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

EX, bit [16]

Export supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EX	Meaning
0b0	PMCR_EL0.X is RES0.
0b1	PMCR_EL0.X is read/write.

Access to this field is RO.

CCD, bit [15]

Cycle counter has prescale.

In Armv8-R AArch32 this is RES1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCD	Meaning
0b0	PMCR_EL0.D is RES0.
0b1	PMCR_EL0.D is read/write.

Access to this field is RO.

CC, bit [14]

Dedicated cycle counter (counter 31) supported.

Reads as 0b1

Access to this field is RO.

SIZE, bits [13:8]

Size of counters, minus one. This field defines the size of the largest counter implemented by the Performance Monitors Unit.

From Armv8.0, the largest counter is 64-bits, so the value of this field is 0b111111.

This field is used by software to determine the spacing of the counters in the memory-map. From Armv8.0, the counters are a doubleword-aligned addresses.

Reads as 0b111111

Access to this field is RO.

N, bits [7:0]

Number of counters accessible, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

N	Meaning
0x00	Only PMCCNTR_EL0 accessible.
0x01..0x1F	Number of counters accessible, 2 to 32, minus one.

All other values are reserved.

The count includes the cycle counter, [PMCCNTR_EL0](#).

When the Effective value of [HDCR.EPMAD](#) is 1, this field reads as the value of [HDCR.HPMN](#).

For example, if [PMCFGR.N](#) == 0x07, then:

- There are eight counters in total.
- This comprises 7 event counters and the cycle counter.

The maximum number of event counters is 31.

Access to this field is RO.

Accessing PMCFGR

Note

[AllowExternalPMUAccess\(\)](#) has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMuV3_EXT32 is implemented

[31:0] Accessible at offset 0xE00 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

F4.3.7 PMCIDR0, Performance Monitors Component Identification Register 0

The PMCIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCIDR0 are RES0.

Attributes

PMCIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

When FEAT_PMU_PIDR is implemented:

Preamble.

Reads as 0x0D

Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing PMCIDR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFF0 from PMU

Accesses to this register are RO.

F4.3.8 PMCIDR1, Performance Monitors Component Identification Register 1

The PMCIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.
For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCIDR1 are RES0.

Attributes

PMCIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

When FEAT_PMU_PIDR is implemented:
Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

PRMBL_1, bits [3:0]

When FEAT_PMU_PIDR is implemented:

Preamble.

Reads as 0b0000

Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing PMCIDR1

Accesses to this register use the following encodings:

When FEAT_PMuV3_EXT is implemented

Accessible at offset 0xFF4 from PMU

Accesses to this register are RO.

F4.3.9 PMCIDR2, Performance Monitors Component Identification Register 2

The PMCIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCIDR2 are RES0.

Attributes

PMCIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

When FEAT_PMU_PIDR is implemented:

Preamble.

Reads as 0x05

Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing PMCIDR2

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFF8 from PMU

Accesses to this register are RO.

F4.3.10 PMCIDR3, Performance Monitors Component Identification Register 3

The PMCIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCIDR3 are RES0.

Attributes

PMCIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

When FEAT_PMU_PIDR is implemented:

Preamble.

Reads as 0xB1

Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing PMCIDR3

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFFC from PMU

Accesses to this register are RO.

F4.3.11 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear Register

The PMCNTENCLR_EL0 characteristics are:

Purpose

Allows software to disable the following counters:

- The cycle counter [PMCCNTR_EL0](#).
- The event counters [PMEVCNTR<n>_EL0](#).

Reading from this register shows which counters are enabled.

Configuration

PMCNTENCLR_EL0 is in the Core power domain.

External register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMCNTENCLR](#)[31:0].

External register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMCNTENSET](#)[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR_EL0 are RES0.

Attributes

PMCNTENCLR_EL0 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

[PMCCNTR_EL0](#) disable. On writes, allows software to disable [PMCCNTR_EL0](#). On reads, returns the [PMCCNTR_EL0](#) enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m>, bits [m], for m = 30 to 0

[PMEVCNTR<m>_EL0](#) disable. On writes, allows software to disable [PMEVCNTR<m>_EL0](#). On reads, returns the [PMEVCNTR<m>_EL0](#) enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{NUM_PMU_COUNTERS}$, access to this field is RAZ/WI.
- When `SoftwareLockStatus(addrdesc)`, access to this field is RO.
- Otherwise, access to this field is WIC.

Accessing PMCNTENCLR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xC20 from PMU

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `!AllowExternalPMUAccess(addrdesc)`, accesses to this register generate an error response.
- When `OSLockStatus()`, accesses to this register generate an error response.
- When `SoftwareLockStatus(addrdesc)`, accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.12 PMCNTESET_EL0, Performance Monitors Count Enable Set Register

The PMCNTESET_EL0 characteristics are:

Purpose

Allows software to enable the following counters:

- The cycle counter [PMCCNTR_EL0](#).
- The event counters [PMEVCNTR<n>_EL0](#).

Reading from this register shows which counters are enabled.

Configuration

PMCNTESET_EL0 is in the Core power domain.

External register PMCNTESET_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMCNTECLR](#)[31:0].

External register PMCNTESET_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMCNTESET](#)[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTESET_EL0 are RES0.

Attributes

PMCNTESET_EL0 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

[PMCCNTR_EL0](#) enable. On writes, allows software to enable [PMCCNTR_EL0](#). On reads, returns the [PMCCNTR_EL0](#) enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is WIS.

P<m>, bits [m], for m = 30 to 0

[PMEVCNTR<m>_EL0](#) enable. On writes, allows software to enable [PMEVCNTR<m>_EL0](#). On reads, returns the [PMEVCNTR<m>_EL0](#) enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{NUM_PMU_COUNTERS}$, access to this field is RAZ/WI.
- When `SoftwareLockStatus(addrdesc)`, access to this field is RO.
- Otherwise, access to this field is WIS.

Accessing PMCNTENSET_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xC00 from PMU

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `!AllowExternalPMUAccess(addrdesc)`, accesses to this register generate an error response.
- When `OSLockStatus()`, accesses to this register generate an error response.
- When `SoftwareLockStatus(addrdesc)`, accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.13 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Configures and controls the Performance Monitors counters.

Configuration

PMCR_EL0 is in the Core power domain.

This register is only partially mapped to the internal PMCR System register. An external agent must use other means to discover the information held in PMCR[31:11], such as accessing PMCFGR and the ID registers.

External register PMCR_EL0 bits [10:0] are architecturally mapped to AArch32 System register PMCR[10:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCR_EL0 are RES0.

Attributes

PMCR_EL0 is a 32-bit register.

Field descriptions

31	11	10	7	6	5	4	3	2	1	0
RAZ/WI										
RES0										
LC										
DP										
X										
D										
C										
P										
E										

Bits [31:11]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

Bits [10:7]

Reserved, RES0.

LC, bit [6]

Long cycle counter enable. Determines when unsigned overflow is recorded by PMOVSLR_EL0.C.

LC	Meaning
0b0	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].
0b1	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

Arm deprecates use of PMCR_EL0.LC = 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DP, bit [5]

Disable cycle counter when event counting is prohibited.

DP	Meaning
0b0	Cycle counting by PMCCNTR_ELO is not affected by this mechanism.
0b1	Cycle counting by PMCCNTR_ELO is disabled in prohibited regions:

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited. For more information about event counter ranges, see [HDCR.HPMN](#).

For more information, see ‘[Prohibiting event and cycle counting](#)’.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
0b0	Do not export events.
0b1	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If [FEAT_ETMv4](#) is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

Clock divider.

D	Meaning
0b0	When enabled, PMCCNTR_ELO counts every clock cycle.

0b1	When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.
-----	--

If the Effective value of PMCR_EL0.LC is 1, then this field is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
0b0	No action.
0b1	Reset PMCCNTR_EL0 to zero.

Note
Resetting [PMCCNTR_EL0](#) does not change the cycle counter overflow field. The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

P	Meaning
0b0	No action.
0b1	Reset all event counters PMEVCNTR<n>_EL0 to zero.

Writes to this field do not affect the cycle counter [PMCCNTR_EL0](#).

Note
Resetting the event counters does not change the event counter overflow fields.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0 .

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see [HDCR.HPMN](#).
- The cycle counter [PMCCNTR_EL0](#).

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PMCR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xE04 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.14 PMDEVAFF0, Performance Monitors Device Affinity register 0

The PMDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

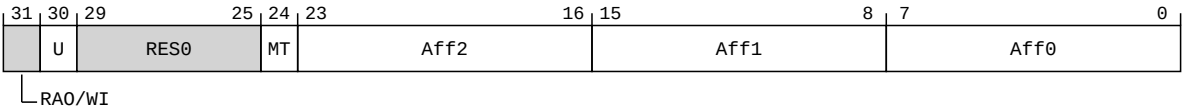
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMuV3_EXT32 is implemented.

Attributes

PMDEVAFF0 is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the [MPIDR](#).{Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xFA8 from PMU

Accesses to this register are RO.

F4.3.15 PMDEVAFF1, Performance Monitors Device Affinity register 1

The PMDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented.

Attributes

PMDEVAFF1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of [PMDEVAFF0.Aff0](#) for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xFAC from PMU

Accesses to this register are RO.

F4.3.16 PMDEVARCH, Performance Monitors Device Architecture register

The PMDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the Performance Monitor component.

Configuration

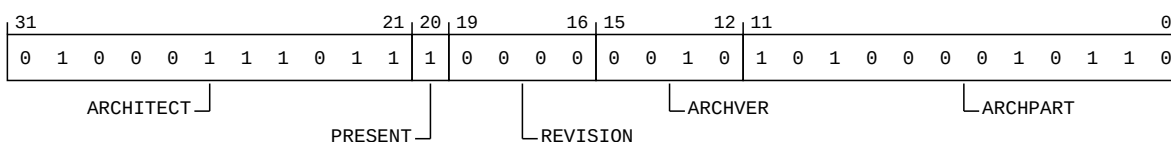
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMuV3_EXT is implemented. Otherwise, direct accesses to PMDEVARCH are RES0.

Attributes

PMDEVARCH is a 32-bit register.

Field descriptions

**ARCHITECT, bits [31:21]**

Defines the architect of the component. For Performance Monitors, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, 0b0100.

Bits [27:21] are the JEP106 identification code, 0b0111011.

Reads as 0b01000111011

Access to this field is RO.

PRESENT, bit [20]

DEVARCH presence. Indicates that the PMDEVARCH register provides architecture information.

Reads as 0b1

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For Performance Monitors, the revision defined by Armv8 is 0x0.

All other values are reserved.

Reads as 0b0000

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0010	Performance Monitors Extension version 3, PMUv3.

All other values are reserved.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12].

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

ARCHPART	Meaning
0xA16	Armv8-A PE performance monitors, including the 32-bit programmers' model extension.

[FEAT_PMUv3_EXT32](#) implements the functionality described by the value 0xA16.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHPART is PMDEVARCH.ARCHID[11:0].

Access to this field is RO.

Accessing PMDEVARCH

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFBC from PMU

Accesses to this register are RO.

F4.3.17 PMDEVTYPE, Performance Monitors Device Type register

The PMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE’s performance monitor interface.

Configuration

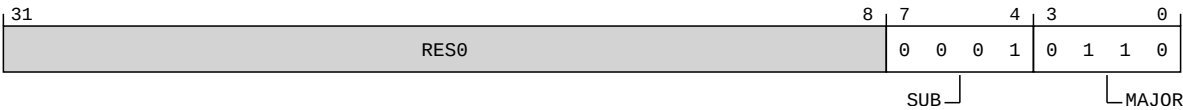
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMU_PIDR is implemented. Otherwise, direct accesses to PMDEVTYPE are RES0.

Attributes

PMDEVTYPE is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as 0b0001

Access to this field is RO.

MAJOR, bits [3:0]

Major type.

MAJOR	Meaning
0b0110	Performance monitor component.

FEAT_PMUv3 implements the functionality described by the value 0b0110.

Access to this field is RO.

Accessing PMDEVTYPE

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFCC from PMU

Accesses to this register are RO.

F4.3.18 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

Purpose

Holds event counter <n>, which counts events, where <n> is 0 to 30.

Configuration

PMEVCNTR<n>_EL0 is in the Core power domain.

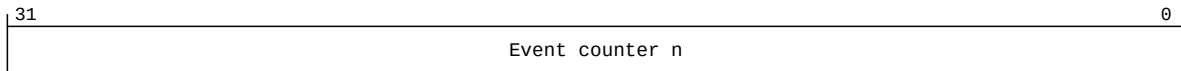
External register PMEVCNTR<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMEVCNTR<n>\[31:0\]](#).

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n>_EL0 are RES0.

Attributes

PMEVCNTR<n>_EL0 is a 32-bit register.

Field descriptions



EVCNT, bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTR<n>_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset $0 \times 000 + (8 * n)$ from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.19 PMEVFILT2R<n>, Performance Monitors Event Filter Registers, n = 0 - 63

The PMEVFILT2R<n> characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED configuration controls for PMU counters.

Each PMEVFILT2R<n> register can provide additional configuration controls for a PMU counter, where:

- For values of n less than 31, if event counter n is implemented, then the controls are for PMU event counter <n>.
- For n equal to 31, the controls are for the cycle counter, [PMCCNTR_ELO](#);
- For all other values of n, PMEVFILT2R<n> is not implemented.

Although this mapping is recommended, it is not required and the function of each register is IMPLEMENTATION DEFINED.

Configuration

PMEVFILT2R<n> is in the Core power domain.

If PMEVFILT2R<n> is not implemented:

- When `IsCorePowered()` && `!DoubleLockStatus()` && `!OSLockStatus()` && `AllowExternalPMUAccess()`, accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

This register is present only when FEAT_PMUv3_EXT is implemented, FEAT_PMUv3 is implemented, and an implementation implements PMEVFILT2R<n>. Otherwise, direct accesses to PMEVFILT2R<n> are RES0.

Attributes

PMEVFILT2R<n> is a 32-bit register.

Field descriptions

31	IMPLEMENTATION DEFINED	0
----	------------------------	---

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing PMEVFILT2R<n>

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset $0 \times 800 + (4 * n)$ from PMU

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `!AllowExternalPMUAccess(addrdesc)`, accesses to this register generate an error response.
- When `OSLockStatus()`, accesses to this register generate an error response.
- When `SoftwareLockStatus(addrdesc)`, accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.20 **PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30**

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configuration

PMEVTYPER<n>_EL0 is in the Core power domain.

If event counter n is not implemented:

- When `IsCorePowered()` && `!DoubleLockStatus()` && `!OSLockStatus()` && `AllowExternalPMUAccess()`, accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

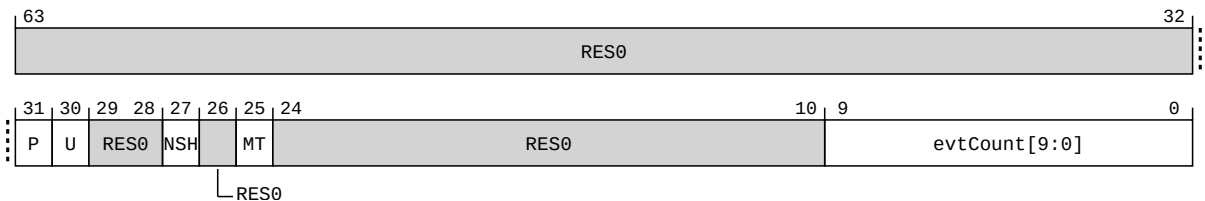
External register PMEVTYPER<n>_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMEVTYPER<n>\[31:0\]](#).

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are RES0.

Attributes

PMEVTYPER<n>_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting events in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL1.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL1 is further controlled by PMEVTYPER<n>_EL0.NSK.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting events in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL0.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL0 is further controlled by PMEVTYPER<n>_EL0.NSU.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 filtering. Controls counting events in EL2.

NSH	Meaning
0b0	The PE does not count events in EL2.
0b1	This mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [26]

Reserved, RES0.

MT, bit [25]

When an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

MT	Meaning
0b0	Count events only on controlling PE.
0b1	Count events from any PE with the same affinity at level 1 and above as this PE.

Unless otherwise stated:

- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.

- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycles when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event.

The second condition is any condition in addition to this.

For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache.

For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

See ‘Multithreaded implementations’ and ‘Cycle event counting in multithreaded implementations’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:10]

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMU.PMEVCNTR<n>_EL0.

The ranges of event numbers allocated to each type of event are shown in ‘Allocation of the PMU event number space’.

If PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range 0x0000 to 0x003F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<n>_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0x400 + (4 * n) from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.21 PMIIDR, Performance Monitors Implementation Identification Register

The PMIIDR characteristics are:

Purpose

Provides discovery information about the Performance Monitor component.

Configuration

This register is present only when FEAT_PMUv3_EXT32 is implemented and an implementation implements PMIIDR. Otherwise, direct accesses to PMIIDR are RES0.

Attributes

PMIIDR is a 32-bit register.

Field descriptions

31	20	19	16	15	12	11	0
ProductID		Variant		Revision		Implementer	

ProductID, bits [31:20]

Part number, bits [11:0]. The part number is selected by the designer of the component.

Matches the part number represented in the Peripheral ID registers PMPIDRn, if those registers are present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by PMIIDR.ProductID, or the major revision of the component.

When defining a major revision, PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the most significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If this field is increased, PMIIDR.Revision should be set to 0b0000.

Arm recommends this field matches the [PMPIDR2.REVISION](#) field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [15:12]

Component minor revision.

PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the least significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If PMIIDR.Variant field is increased, this field should be set to 0b0000, otherwise the value in this field should be increased.

Arm recommends this field matches the [PMPIDR3.REVAND](#) field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the PMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for PMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as 0x43B.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If [PMPIDR4](#) is implemented, [PMPIDR4.DES_2](#) matches bits [11:8] of this field.

If [PMPIDR2](#) is implemented, [PMPIDR2.DES_1](#) matches bits [6:4] of this field.

If [PMPIDR1](#) is implemented, [PMPIDR1.DES_0](#) matches bits [3:0] of this field.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMIIDR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xE08 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

F4.3.22 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear Register

The PMINTENCLR_EL1 characteristics are:

Purpose

Allows software to disable the generation of interrupt requests on overflows from the following counters:

- The cycle counter [PMCCNTR_EL0](#).
- The event counters [PMEVCNTR<n>_EL0](#).

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

PMINTENCLR_EL1 is in the Core power domain.

External register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register [PMINTENCLR](#)[31:0].

External register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to AArch32 System register [PMINTENSET](#)[31:0].

This register is present only when FEAT_PMuV3_EXT is implemented and FEAT_PMuV3 is implemented. Otherwise, direct accesses to PMINTENCLR_EL1 are RES0.

Attributes

PMINTENCLR_EL1 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request on unsigned overflow of [PMCCNTR_EL0](#) disable. On writes, allows software to disable the interrupt request on unsigned overflow of [PMCCNTR_EL0](#). On reads, returns the interrupt request on unsigned overflow of [PMCCNTR_EL0](#) enable status.

C	Meaning
0b0	Interrupt request on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is WIC.

P<m> , bits [m], for m = 30 to 0

Interrupt request on unsigned overflow of [PMEVCNTR<m>_EL0](#) disable. On writes, allows software to disable the interrupt request on unsigned overflow of [PMEVCNTR<m>_EL0](#). On reads, returns the interrupt request on unsigned overflow of [PMEVCNTR<m>_EL0](#) enable status.

P<m>	Meaning
0b0	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{NUM_PMU_COUNTERS}$, access to this field is RAZ/WI.
- When `SoftwareLockStatus(addrdesc)`, access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMINTENCLR_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xC60 from PMU

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `!AllowExternalPMUAccess(addrdesc)`, accesses to this register generate an error response.
- When `OSLockStatus()`, accesses to this register generate an error response.
- When `SoftwareLockStatus(addrdesc)`, accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.23 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register

The PMINTENSET_EL1 characteristics are:

Purpose

Allows software to enable the generation of interrupt requests on overflows from the following counters:

- The cycle counter [PMCCNTR_EL0](#).
- The event counters [PMEVCNTR<n>_EL0](#).

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

PMINTENSET_EL1 is in the Core power domain.

External register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register [PMINTENSET](#)[31:0].

External register PMINTENSET_EL1 bits [31:0] are architecturally mapped to AArch32 System register [PMINTENCLR](#)[31:0].

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET_EL1 are RES0.

Attributes

PMINTENSET_EL1 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request on unsigned overflow of [PMCCNTR_EL0](#) enable. On writes, allows software to enable the interrupt request on unsigned overflow of [PMCCNTR_EL0](#). On reads, returns the interrupt request on unsigned overflow of [PMCCNTR_EL0](#) enable status.

C	Meaning
0b0	Interrupt request on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is WIS.

P<m> , bits [m], for m = 30 to 0

Interrupt request on unsigned overflow of [PMEVCNTR<m>_EL0](#) enable. On writes, allows software to enable the interrupt request on unsigned overflow of [PMEVCNTR<m>_EL0](#). On reads, returns the interrupt request on unsigned overflow of [PMEVCNTR<m>_EL0](#) enable status.

P<m>	Meaning
0b0	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{NUM_PMU_COUNTERS}$, access to this field is RAZ/WI.
- When `SoftwareLockStatus(addrdesc)`, access to this field is RO.
- Otherwise, access to this field is WIS.

Accessing PMINTENSET_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0xC40` from PMU

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `!AllowExternalPMUAccess(addrdesc)`, accesses to this register generate an error response.
- When `OSLockStatus()`, accesses to this register generate an error response.
- When `SoftwareLockStatus(addrdesc)`, accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.24 PMITCTRL, Performance Monitors Integration mode Control register

The PMITCTRL characteristics are:

Purpose

Enables the Performance Monitors to switch from default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

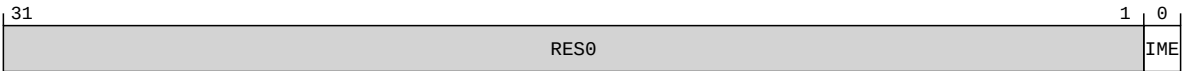
Configuration

This register is present only when FEAT_PMUv3_EXT is implemented, FEAT_PMUv3 is implemented, and an implementation implements PMITCTRL. Otherwise, direct accesses to PMITCTRL are RES0.

Attributes

PMITCTRL is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
 - On a Cold reset, this field resets to 0.
 - On an External debug reset, the value of this field is unchanged.
 - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
 - On a Cold reset, the value of this field is unchanged.
 - On an External debug reset, this field resets to 0.
 - On a Warm reset, the value of this field is unchanged.

Accessing PMITCTRL

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xF00 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.25 PMLAR, Performance Monitors Lock Access Register

The PMLAR characteristics are:

Purpose

Allows or disallows access to the Performance Monitors registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and [PMLSR](#) to check the current status of the lock.

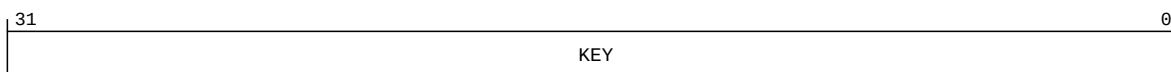
This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMLAR are RES0.

Attributes

PMLAR is a 32-bit register.

Field descriptions

When PMU Software Lock is implemented:

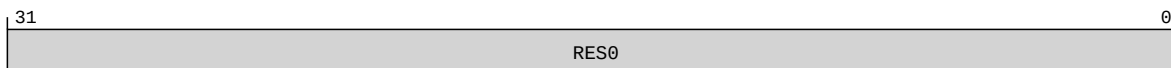


KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:



Otherwise

Bits [31:0]

Reserved, RES0.

Accessing PMLAR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0x`FB0` from PMU

Accesses to this register are WO.

F4.3.26 PMLSR, Performance Monitors Lock Status Register

The PMLSR characteristics are:

Purpose

Indicates the current status of the software lock for Performance Monitors registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses **PMLAR** to set or clear the lock, and **PMLSR** to check the current status of the lock.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMLSR are RES0.

Attributes

PMLSR is a 32-bit register.

Field descriptions

**Bits [31:3]**

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required.

Reads as 0b0

Access to this field is RO.

SLK, bit [1]

When PMU Software Lock is implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when Software Lock is implemented, possible values of this field are:

SLK	Meaning
0b0	Lock clear. Writes are permitted to this component's registers.

0b1

Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a External debug reset, this field resets to '1'.

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
0b0	Software Lock not implemented or not memory-mapped access.
0b1	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing PMLSR

Accesses to this register use the following encodings:

When FEAT_PMuV3_EXT is implemented

Accessible at offset 0xFB4 from PMU

Accesses to this register are RO.

F4.3.27 PMOVSLR_EL0, Performance Monitors Overflow Flag Status Clear register

The PMOVSLR_EL0 characteristics are:

Purpose

Allows software to clear the unsigned overflow flags for the following counters to 0:

- The cycle counter [PMCCNTR_EL0](#).
- The event counters [PMEVCNTR<n>_EL0](#).

Reading from this register shows the current unsigned overflow flag values.

Configuration

PMOVSLR_EL0 is in the Core power domain.

External register PMOVSLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMOVSRR\[31:0\]](#).

External register PMOVSLR_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMOVSSET\[31:0\]](#).

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSLR_EL0 are RES0.

Attributes

PMOVSLR_EL0 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for [PMCCNTR_EL0](#) clear. On writes, allows software to clear the unsigned overflow flag for [PMCCNTR_EL0](#) to 0. On reads, returns the unsigned overflow flag for [PMCCNTR_EL0](#) overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

[PMCR_EL0.LC](#) controls whether an overflow is detected from unsigned overflow of [PMCCNTR_EL0](#)[31:0] or unsigned overflow of [PMCCNTR_EL0](#)[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is WIC.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for [PMEVCNTR<m>_EL0](#) clear. On writes, allows software to clear the unsigned overflow flag for [PMEVCNTR<m>_EL0](#) to 0. On reads, returns the unsigned overflow flag for [PMEVCNTR<m>_EL0](#) overflow status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMOVSLR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xC80 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.28 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register

The PMOVSSET_EL0 characteristics are:

Purpose

Allows software to set the unsigned overflow flags for the following counters to 1:

- The cycle counter [PMCCNTR_EL0](#).
- The event counters [PMEVCNTR<n>_EL0](#).

Reading from this register shows the current unsigned overflow flag values.

Configuration

PMOVSSET_EL0 is in the Core power domain.

External register PMOVSSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMOVSRR\[31:0\]](#).

External register PMOVSSET_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMOVSSET\[31:0\]](#).

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSSET_EL0 are RES0.

Attributes

PMOVSSET_EL0 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for [PMCCNTR_EL0](#) set. On writes, allows software to set the unsigned overflow flag for [PMCCNTR_EL0](#) to 1. On reads, returns the unsigned overflow flag for [PMCCNTR_EL0](#) overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

[PMCR_EL0.LC](#) controls whether an overflow is detected from unsigned overflow of [PMCCNTR_EL0](#)[31:0] or unsigned overflow of [PMCCNTR_EL0](#)[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(addrdesc), access to this field is RO.
- Otherwise, access to this field is WIS.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for [PMEVCNTR<m>_EL0](#) set. On writes, allows software to set the unsigned overflow flag for [PMEVCNTR<m>_EL0](#) to 1. On reads, returns the unsigned overflow flag for [PMEVCNTR<m>_EL0](#) overflow status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When $m \geq \text{NUM_PMU_COUNTERS}$, access to this field is RAZ/WI.
- When `SoftwareLockStatus(addrdesc)`, access to this field is RO.
- Otherwise, access to this field is WIS.

Accessing PMOVSSET_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xCC0 from PMU

- When `DoubleLockStatus()`, or `!IsCorePowered()`, or `!AllowExternalPMUAccess(addrdesc)`, accesses to this register generate an error response.
- When `OSLockStatus()`, accesses to this register generate an error response.
- When `SoftwareLockStatus(addrdesc)`, accesses to this register are RO.
- Otherwise, accesses to this register are RW.

F4.3.29 PMPIDR0, Performance Monitors Peripheral Identification Register 0

The PMPIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMuV3_EXT is implemented and FEAT_PMu_PIDR is implemented. Otherwise, direct accesses to PMPIDR0 are RES0.

Attributes

PMPIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
 - [PMPIDR1.PART_1](#) contains part number bits [11:8].
 - [PMPIDR0.PART_0](#) contains part number bits [7:0].
- For a component with a 16-bit part number:
 - [PMPIDR1.PART_1](#) contains part number bits [15:12].
 - [PMPIDR0.PART_0](#) contains part number bits [11:4].
 - [PMPIDR2.REVISION](#) contains part number bits [3:0].

When a 12-bit part number is used, [PMPIDR2.REVISION](#) indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR0

Accesses to this register use the following encodings:

When FEAT_PMuV3_EXT is implemented

Accessible at offset 0xFE0 from PMU

Accesses to this register are RO.

F4.3.30 PMPIDR1, Performance Monitors Peripheral Identification Register 1

The PMPIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMU_PIDR is implemented. Otherwise, direct accesses to PMPIDR1 are RES0.

Attributes

PMPIDR1 is a 32-bit register.

Field descriptions

31		8	7	4	3	0	
RES0							
DES_0							
PART_1							

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

The JEP106 identification and continuation codes are stored as follows:

- **PMPIDR1.DES_0**: JEP106 identification code bits[3:0].
- **PMPIDR2.DES_1**: JEP106 identification code bits[6:4].
- **PMPIDR4.DES_2**: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value $0 \times 7F$.
- A following 8-bit number, that is not $0 \times 7F$, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code $0 \times 7F \ 0 \times 7F \ 0 \times 7F \ 0 \times 7F \ 0 \times 3B$.

- The continuation code is the number of times $0 \times 7F$ appears before the final number. For example, a component designed by Arm Limited has the code 0×4 .
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code $0 \times 3B$.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, which is selected by the designer of the component and stored as follows:

- For a component with a 12-bit part number:
 - [PMPIDR1](#).PART_1 contains part number bits [11:8].
 - [PMPIDR0](#).PART_0 contains part number bits [7:0].
- For a component with a 16-bit part number:
 - [PMPIDR1](#).PART_1 contains part number bits [15:12].
 - [PMPIDR0](#).PART_0 contains part number bits [11:4].
 - [PMPIDR2](#).REVISION contains part number bits [3:0].

When a 12-bit part number is used, [PMPIDR2](#).REVISION indicates revision information.

The choice of using a 12-bit part number or 16-bit part number is specific to the designer of the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR1

Accesses to this register use the following encodings:

When FEAT_PMuV3_EXT is implemented

Accessible at offset 0xFE4 from PMU

Accesses to this register are RO.

F4.3.31 PMPIDR2, Performance Monitors Peripheral Identification Register 2

The PMPIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

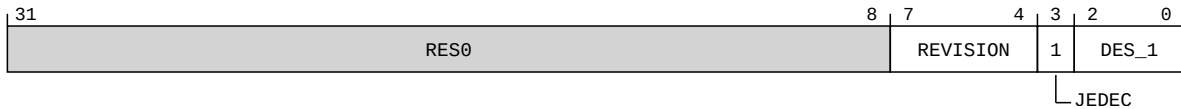
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMU_PIDR is implemented. Otherwise, direct accesses to PMPIDR2 are RES0.

Attributes

PMPIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Indicates either the revision of the component, or a portion of the part number of the component.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- [PMPIDR2.REVISION](#) indicates the 4-bit revision number.
- [PMPIDR3.REVAND](#) indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- [PMPIDR2.REVISION](#) indicates the 4-bit major revision number.
- [PMPIDR3.REVAND](#) indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- [PMPIDR2.REVISION](#) contains part number bits [3:0].
- [PMPIDR3.REVAND](#) indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as 0b1

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

The JEP106 identification and continuation codes are stored as follows:

- **PMPIDR1.DES_0**: JEP106 identification code bits[3:0].
- **PMPIDR2.DES_1**: JEP106 identification code bits[6:4].
- **PMPIDR4.DES_2**: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value 0x7F.
- A following 8-bit number, that is not 0x7F, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code 0x7F 0x7F 0x7F 0x7F 0x3B.

- The continuation code is the number of times 0x7F appears before the final number. For example, a component designed by Arm Limited has the code 0x4.
 - The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code 0x3B.
-

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR2

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFE8 from PMU

Accesses to this register are RO.

F4.3.32 PMPIDR3, Performance Monitors Peripheral Identification Register 3

The PMPIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMU_PIDR is implemented. Otherwise, direct accesses to PMPIDR3 are RES0.

Attributes

PMPIDR3 is a 32-bit register.

Field descriptions

31	8	7	4	3	0	
RES0					REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Indicates either the revision of the component, or whether the component has been modified.

Where the component has a single 4-bit revision number, the revision number is an incremental value starting at zero for the first revision of the component.

Where the component has separate major and minor revision numbers, the major and minor revision numbers are each incremental values starting at zero for the first revision of the component. For each minor revision of the component, the minor revision number increments monotonically. For each major revision of the component, the major revision number increments monotonically and the minor revision begins again at zero.

For a component with a 12-bit part number with a single 4-bit revision number:

- [PMPIDR2.REVISION](#) indicates the 4-bit revision number.
- [PMPIDR3.REVAND](#) indicates component modifications.

For a component with a 12-bit part number with separate major and minor revision numbers:

- [PMPIDR2.REVISION](#) indicates the 4-bit major revision number.
- [PMPIDR3.REVAND](#) indicates the 4-bit minor revision number.

For a component with a 16-bit part number:

- [PMPIDR2.REVISION](#) contains part number bits [3:0].
- [PMPIDR3.REVAND](#) indicates the 4-bit revision number.

The choice of which style of revision information is used is specific to the designer of the component, and might also be specific to each individual component with a different part number.

Where REVAND indicates component modifications, this indicates modifications such as errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field

is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

[PMPIDR3](#).CMOD might also indicate component modifications.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Indicates whether the component has been modified from its original behavior. Examples of modifications include errata fixes or metal fixes after implementation. Usually this value would be zero unless a modification has been performed. If the field is required for indicating component modifications, Arm recommends that component designers ensure that it can be changed by a metal fix, for example by driving it from registers that reset to zero.

A value of 0b0000 means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

[PMPIDR3](#).REVAND might also indicate component modifications.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR3

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFEC from PMU

Accesses to this register are RO.

F4.3.33 PMPIDR4, Performance Monitors Peripheral Identification Register 4

The PMPIDR4 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

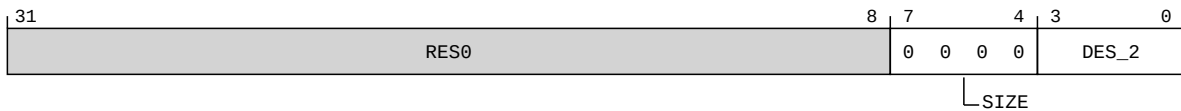
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMU_PIDR is implemented. Otherwise, direct accesses to PMPIDR4 are RES0.

Attributes

PMPIDR4 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. $\log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as 0b0000

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

The JEP106 identification and continuation codes are stored as follows:

- **PMPIDR1.DES_0**: JEP106 identification code bits[3:0].
- **PMPIDR2.DES_1**: JEP106 identification code bits[6:4].
- **PMPIDR4.DES_2**: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC <http://www.jedec.org>.

A JEP106 identification and continuation code takes the following form:

- A sequence of zero or more numbers, all having the value 0x7F.
- A following 8-bit number, that is not 0x7F, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

Note

For example, Arm Limited is assigned the code 0x7F 0x7F 0x7F 0x7F 0x3B.

- The continuation code is the number of times 0x7F appears before the final number. For example, a component designed by Arm Limited has the code 0x4.
 - The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code 0x3B.
-

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR4

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFD0 from PMU

Accesses to this register are RO.

F4.3.34 PMPIDR5, Performance Monitors Peripheral Identification Register 5

The PMPIDR5 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

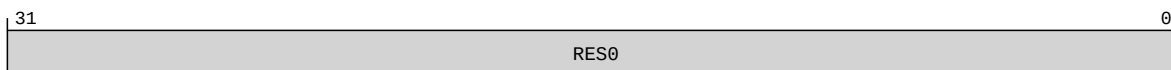
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMU_PIDR is implemented. Otherwise, direct accesses to PMPIDR5 are RES0.

Attributes

PMPIDR5 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing PMPIDR5

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFD4 from PMU

Accesses to this register are RO.

F4.3.35 PMPIDR6, Performance Monitors Peripheral Identification Register 6

The PMPIDR6 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

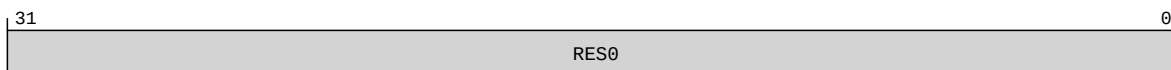
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMU_PIDR is implemented. Otherwise, direct accesses to PMPIDR6 are RES0.

Attributes

PMPIDR6 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing PMPIDR6

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFD8 from PMU

Accesses to this register are RO.

F4.3.36 PMPIDR7, Performance Monitors Peripheral Identification Register 7

The PMPIDR7 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

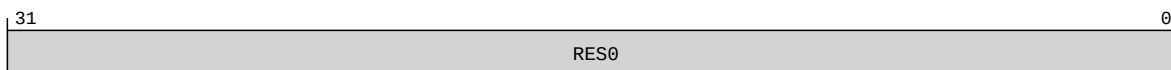
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMU_PIDR is implemented. Otherwise, direct accesses to PMPIDR7 are RES0.

Attributes

PMPIDR7 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing PMPIDR7

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFDC from PMU

Accesses to this register are RO.

F4.3.37 PMSWINC_EL0, Performance Monitors Software Increment Register

The PMSWINC_EL0 characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event 0x00. For more information, see ‘SW_INCR’.

Configuration

PMSWINC_EL0 is in the Core power domain.

If this register is implemented, use of it is deprecated.

If 1 is written to bit [n] from the external debug interface, it is CONSTRAINED UNPREDICTABLE whether or not a SW_INCR event is created for counter n. This is consistent with not implementing the register in the external debug interface.

When FEAT_PMUv3_EXT32 is implemented, External register PMSWINC_EL0 bits [31:0] are architecturally mapped to AArch32 System register [PMSWINC](#)[31:0].

This register is present only when FEAT_PMUv3_EXT32 is implemented and an implementation implements PMSWINC_EL0.

Attributes

PMSWINC_EL0 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

RES0

Bit [31]

Reserved, RES0.

P<m> , bits [m], for m = 30 to 0

Event counter software increment bit for [PMEVCNTR<m>_EL0](#).

P<m>	Meaning
0b0	No action. The write to this bit is ignored.
0b1	It is CONSTRAINED UNPREDICTABLE whether a SW_INCR event is generated for event counter m.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(addrdesc), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Accessing PMSWINC_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xCA0 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(addrdesc), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

F4.4 Generic Timer memory-mapped registers overview

The Generic Timer memory-mapped registers are implemented as multiple register frames, with each register frame having its own base address, as follows:

- A single CNTCTLBase register frame, at base address CNTCTLBase.
- Between one and seven CNTBaseN register frames, each with its own base address CNTBaseN.
- For each CNTBaseN register frame, if required, a CNTEL0BaseN register frame, at base address CNTEL0BaseN, that provides an EL0 view of the CNTBaseN register frame.

For more information, see:

- [Memory-mapped timer components](#).
- [The CNTBaseN and CNTEL0BaseN frames](#). This section includes the memory map of the CNTBaseN and CNTEL0BaseN register frames.
- [The CNTCTLBase frame](#). This section includes the memory map of the CNTCTLBase register frame.

Note

Providing a complete set of features in a system level implementation gives an implementation example for a system level implementation of the Generic Timer.

F4.5 Generic Timer memory-mapped register descriptions

This section describes the Generic Timer registers. [Generic Timer memory-mapped registers overview](#) gives an overview of these registers, and includes links to their memory maps.

F4.5.1 CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7

The CNTACR<n> characteristics are:

Purpose

Provides top-level access controls for the elements of a timer frame. CNTACR<n> provides the controls for frame CNTBaseN.

In addition to the CNTACR<n> control:

- CNTNSAR controls whether CNTACR<n> is accessible by Non-secure accesses.
- If frame CNTEL0BaseN is implemented, the CNTEL0ACR in frame CNTBaseN provides additional control of accesses to frame CNTEL0BaseN.

Configuration

It is IMPLEMENTATION DEFINED whether CNTACR<n> is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Implemented only if the value of CNTTIDR.Frame<n> is 1.

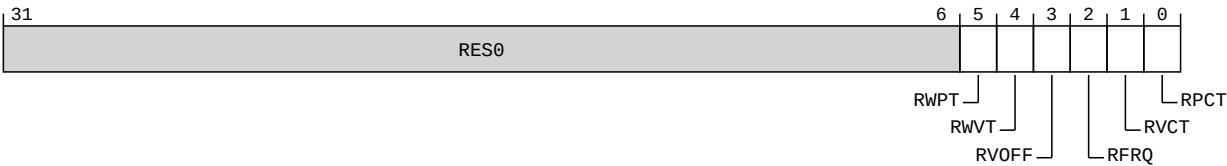
An implementation of the counters might not provide configurable access to some or all of the features. In this case, the associated field in the CNTACR<n> register is:

- RAZ/WI if access is always denied.
- RAO/WI if access is always permitted.

Attributes

CNTACR<n> is a 32-bit register.

Field descriptions



Bits [31:6]

Reserved, RES0.

RWPT, bit [5]

Read/write access to the EL1 Physical Timer registers CNTP_CVAL, CNTP_TVAL, and CNTP_CTL, in frame <n>.

RWPT	Meaning
0b0	No access to the EL1 Physical Timer registers in frame <n>. The registers are RES0.
0b1	Read/write access to the EL1 Physical Timer registers in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RWVT, bit [4]

Read/write access to the Virtual Timer register [CNTV_CVAL](#), [CNTV_TVAL](#), and [CNTV_CTL](#), in frame <n>.

RWVT	Meaning
0b0	No access to the Virtual Timer registers in frame <n>. The registers are RES0.
0b1	Read/write access to the Virtual Timer registers in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVOFF, bit [3]

Read-only access to [CNTVOFF](#), in frame <n>.

RVOFF	Meaning
0b0	No access to CNTVOFF in frame <n>. The register is RES0.
0b1	Read-only access to CNTVOFF in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RFRQ, bit [2]

Read-only access to [CNTFRQ](#), in frame <n>.

RFRQ	Meaning
0b0	No access to CNTFRQ in frame <n>. The register is RES0.
0b1	Read-only access to CNTFRQ in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVCT, bit [1]

Read-only access to [CNTVCT](#), in frame <n>.

RVCT	Meaning
0b0	No access to CNTVCT in frame <n>. The register is RES0.
0b1	Read-only access to CNTVCT in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RPCT, bit [0]

Read-only access to CNTPCT, in frame <n>.

RPCT	Meaning
0b0	No access to CNTPCT in frame <n>. The register is RES0.
0b1	Read-only access to CNTPCT in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTACR<n>

In a system that recognizes two Security states:

- CNTACR<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTACR<n> is accessible by Non-secure accesses.

CNTACR<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	$0 \times 040 + (4 * n)$	CNTACR<n>

Accesses to this register are RW.

F4.5.2 CNTCR, Counter Control Register

The CNTCR characteristics are:

Purpose

Enables the counter, controls the [counter resolution](#), and controls counter behavior during debug.

Configuration

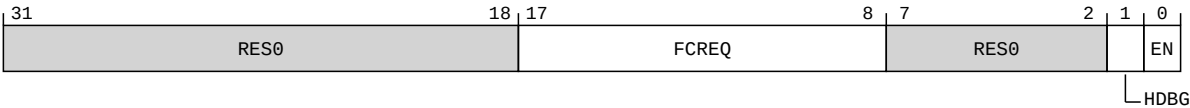
It is IMPLEMENTATION DEFINED whether CNTCR is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

Attributes

CNTCR is a 32-bit register.

Field descriptions



Bits [31:18]

Reserved, RES0.

FCREQ, bits [17:8]

Frequency change request. Indicates the number of the entry in the Frequency modes table to select.

Selecting an unimplemented entry, or an entry that contains 0, has no effect on the counter.

The maximum number of entries in the Frequency modes table is IMPLEMENTATION DEFINED up to a maximum of 1004 entries, see [‘The Frequency modes table’](#). An implementation is only required to implement an FCREQ field that can hold values from 0 to the highest supported Frequency modes table entry. Any unrequired most-significant bits of FCREQ can be implemented as RES0.

The reset behavior of this field is:

- On a Timer reset, this field resets to the expression 0x0.

Bits [7:2]

Reserved, RES0.

HDBG, bit [1]

Halt-on-debug. Controls whether a Halt-on-debug signal halts the system counter:

HDBG	Meaning
0b0	System counter ignores Halt-on-debug.
0b1	Asserted Halt-on-debug signal halts system counter update.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Enables the counter:

EN	Meaning
0b0	System counter disabled.
0b1	System counter enabled.

The reset behavior of this field is:

- On a Timer reset, this field resets to '0'.

Accessing CNTCR

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTCR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	0x000	CNTCR

Accesses to this register are RW.

F4.5.3 CNTCV, Counter Count Value register

The CNTCV characteristics are:

Purpose

Indicates the current count value.

Configuration

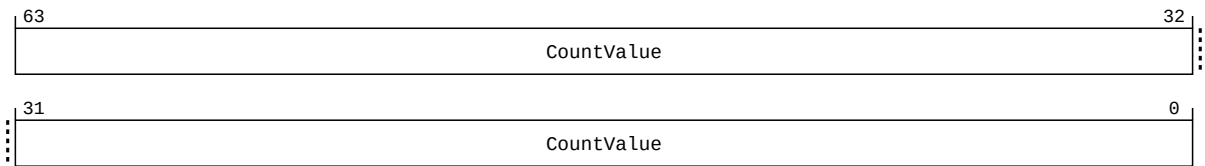
It is IMPLEMENTATION DEFINED whether CNTCV is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTCV is a 64-bit register.

Field descriptions



CountValue, bits [63:0]

Indicates the counter value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTCV

Frame	Accessibility
CNTControlBase	RW
CNTReadBase	RO

A write to CNTCV must be visible in the CNTPCT register of each running processor in a finite time.

For the instance of the register in the CNTControlBase frame:

- In a system that supports Secure and Non-secure physical address spaces, this register is implemented only in the Secure physical address space.
- If the counter is enabled, the effect of writing to the register is UNKNOWN.

For the instance of the register in the CNTReadBase frame, this register is accessible in all physical address spaces.

In an implementation that supports 64-bit atomic memory accesses, this register must be accessible using a 64-bit atomic access.

CNTCV can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTControlBase	0x008	CNTCV	63:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTReadBase	0x000	CNTCV	63:0

Accesses to this register are RO.

F4.5.4 CNTEL0ACR, Counter-timer EL0 Access Control Register

The CNTEL0ACR characteristics are:

Purpose

An implementation of CNTEL0ACR in the frame at CNTBaseN controls whether the [CNTPCT](#), [CNTVCT](#), [CNTFRQ](#), EL1 Physical Timer, and Virtual Timer registers are visible in the frame at CNTEL0BaseN.

Configuration

It is IMPLEMENTATION DEFINED whether CNTEL0ACR is implemented in the Core power domain or in the Debug power domain

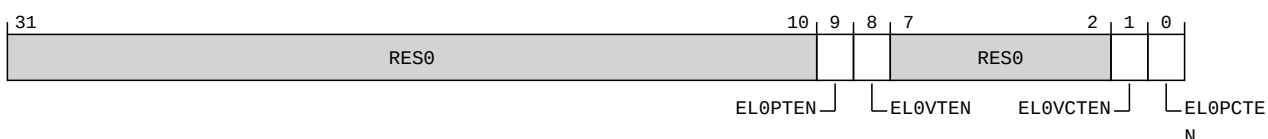
For more information, see ‘[Power and reset domains for the system level implementation of the Generic Timer](#)’.

Implementation of this register is OPTIONAL.

Attributes

CNTEL0ACR is a 32-bit register.

Field descriptions



Bits [31:10]

Reserved, RES0.

EL0PTEN, bit [9]

Second view read/write access control for the EL1 Physical Timer registers. This bit controls whether the [CNTP_CVAL](#), [CNTP_TVAL](#), and [CNTP_CTL](#) registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.

EL0PTEN	Meaning
0b0	No access. Registers are RES0 in the second view.
0b1	Access permitted. If the registers are accessible in the current frame, then they are accessible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

Second view read/write access control for the Virtual Timer registers. This bit controls whether the [CNTV_CVAL](#), [CNTV_TVAL](#), and [CNTV_CTL](#) registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.

EL0VTEN	Meaning
0b0	No access. Registers are RES0 in the second view.
0b1	Access permitted. If the registers are accessible in the current frame, then they are accessible in the second view.

The definition of this bit means that, if the Virtual Timer registers are not implemented in the current CNTBaseN frame, then the Virtual Timer register addresses are RES0 in the corresponding CNTEL0BaseN frame, regardless of the value of this bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bits [7:2]

Reserved, RES0.

EL0VCTEN, bit [1]

Second view read access control for [CNTVCT](#) and [CNTFRQ](#).

EL0VCTEN	Meaning
0b0	CNTVCT is not visible in the second view. If EL0PCTEN is set to 0, CNTFRQ is not visible in the second view.
0b1	Access permitted. If CNTVCT and CNTFRQ are visible in the current frame, then they are visible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Second view read access control for [CNTPCT](#) and [CNTFRQ](#).

EL0PCTEN	Meaning
0b0	CNTPCT is not visible in the second view. If EL0VCTEN is set to 0, CNTFRQ is not visible in the second view.
0b1	Access permitted. If CNTPCT and CNTFRQ are visible in the current frame, then they are visible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTEL0ACR

CNTEL0ACR can be implemented in any implemented CNTBaseN frame.

‘[CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames](#)’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

If CNTEL0ACR is not implemented in an implemented CNTBaseN frame:

- The register location in that frame is RAZ/WI.
- If the corresponding CNTEL0BaseN frame is implemented, the registers [CNTFRQ](#), [CNTP_CTL](#), [CNTP_CVAL](#), [CNTP_TVAL](#), [CNTPCT](#), [CNTV_CTL](#), [CNTV_CVAL](#), [CNTV_TVAL](#), and [CNTVCT](#) are not visible and accesses are RAZ/WI in that frame.

CNTEL0ACR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x014	CNTEL0ACR

Accesses to this register are RW.

F4.5.5 CNTFID0, Counter Frequency ID

The CNTFID0 characteristics are:

Purpose

Indicates the base frequency of the system counter.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFID0 is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0. For more information, see [‘The Frequency modes table’](#).

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

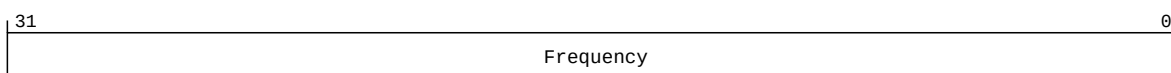
Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID0 is a 32-bit register.

Field descriptions



Frequency, bits [31:0]

The base frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFID0

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTFID0 can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	0x020	CNTFID0

Accesses to this register are IMPLEMENTATION DEFINED.

F4.5.6 CNTFID<n>, Counter Frequency IDs, $n > 0$, $n = 1 - 1003$

The CNTFID<n> characteristics are:

Purpose

Indicates alternative system counter frequencies.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFID<n> is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0, see [‘The Frequency modes table’](#).

The number of CNTFID<n> registers is IMPLEMENTATION DEFINED, and the only required CNTFID<n> register is CNTFID0.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED up to this limit. For an implementation that includes registers in the IMPLEMENTATION DEFINED register space $0x0C0-0x0FC$, the maximum number of entries in the Frequency modes table is 40, including the zero-word end marker.

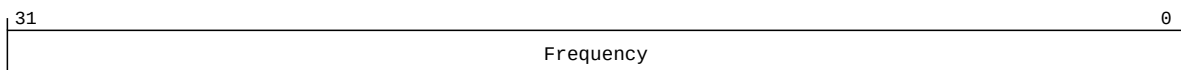
Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID<n> is a 32-bit register.

Field descriptions



Frequency, bits [31:0]

A system counter frequency, in Hz. Must be an exact divisor of the base frequency. Arm strongly recommends that all frequency values in the Frequency modes table are integer power-of-two divisors of the base frequency.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is increased by the ratio of the base frequency and the selected frequency.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFID<n>

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes these registers, is implemented only in the Secure physical address space.

CNTFID<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	$0 \times 020 + (4 * n)$	CNTFID<n>

Accesses to this register are IMPLEMENTATION DEFINED.

F4.5.7 CNTFRQ, Counter-timer Frequency

The CNTFRQ characteristics are:

Purpose

This register is provided so that software can discover the effective frequency of the system counter. The instance of the register in the CNTCTLBase frame must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configuration

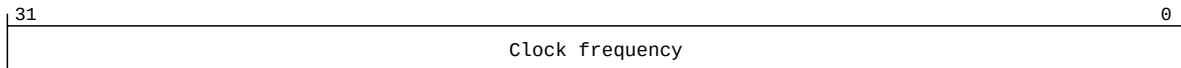
It is IMPLEMENTATION DEFINED whether CNTFRQ is implemented in the Core power domain or in the Debug power domain

For more information see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

Attributes

CNTFRQ is a 32-bit register.

Field descriptions



ClockFreq, bits [31:0]

Clock frequency. Indicates the effective frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFRQ

CNTFRQ must be implemented as an RW register in the CNTCTLBase frame.

In a system that recognizes two Security states, the instance of the register in the CNTCTLBase frame is only accessible by Secure accesses.

CNTFRQ can be implemented as a RO register in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

[‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’](#) describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTFRQ is accessible in that frame, as a RO register, if the value of [CNTACR<n>.RFRQ](#) is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTFRQ is accessible as a RO register in that frame if both:

- CNTFRQ is accessible in the corresponding CNTBaseN frame.
- Either the value of CNTEL0ACR.EL0VCTEN is 1 or the value of CNTEL0ACR.EL0PCTEN is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

CNTFRQ can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x010	CNTFRQ

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x010	CNTFRQ

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTCTLBase	0x000	CNTFRQ

Accesses to this register are RO.

F4.5.8 CNTNSAR, Counter-timer Non-secure Access Register

The CNTNSAR characteristics are:

Purpose

Provides the highest-level control of whether frames CNTBaseN and CNTEL0BaseN are accessible by Non-secure accesses.

Configuration

It is IMPLEMENTATION DEFINED whether CNTNSAR is implemented in the Core power domain or in the Debug power domain

For more information, see ‘[Power and reset domains for the system level implementation of the Generic Timer](#)’.

Attributes

CNTNSAR is a 32-bit register.

Field descriptions

**Bits [31:8]**

Reserved, RES0.

NS<n> , bits [n], for n = 7 to 0

Non-secure access to frame n.

NS<n>	Meaning
0b0	Secure access only. Behaves as RES0 to Non-secure accesses.
0b1	Secure and Non-secure accesses permitted.

This bit also determines whether, in the CNTCTLBase frame, CNTACR<n> and CNTVOFF<n> are accessible to Non-secure accesses.

If frame CNTBase<n>:

- Is not implemented, then NS<n> is RES0.
- Is not Configurable access, and is accessible only by Secure accesses, then NS<n> is RES0.
- Is not Configurable access, and is accessible by both Secure and Non-secure accesses, then NS<n> is RES1.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTNSAR

In a system that recognizes two Security states, this register is only accessible by Secure accesses.

CNTNSAR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	0x004	CNTNSAR

Accesses to this register are RW.

F4.5.9 CNTPCT, Counter-timer Physical Count

The CNTPCT characteristics are:

Purpose

Holds the 64-bit physical count value.

Configuration

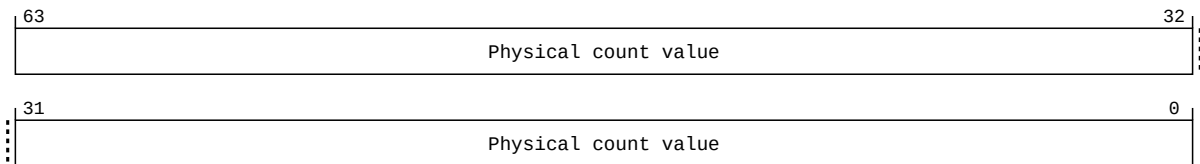
It is IMPLEMENTATION DEFINED whether CNTPCT is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

Attributes

CNPCT is a 64-bit register.

Field descriptions



PhysicalCount, bits [63:0]

Physical count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCT

CNPCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

[‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’](#) describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNPCT is accessible in that frame, as a RO register, if the value of [CNTACR<n>.RPCT](#) is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNPCT is accessible in that frame if both:
 - CNPCT is accessible in the corresponding CNTBaseN frame.
 - The value of [CNTEL0ACR.EL0PCTEN](#) is 1.

- Otherwise, the CNTPCT address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTPCT register must be accessible as an atomic 64-bit value.

CNTPCT can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x000	CNTPCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x004	CNTPCT	63:32

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x000	CNTPCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x004	CNTPCT	63:32

Accesses to this register are RO.

F4.5.10 CNTP_CTL, Counter-timer Physical Timer Control

The CNTP_CTL characteristics are:

Purpose

Control register for the EL1 physical timer.

Configuration

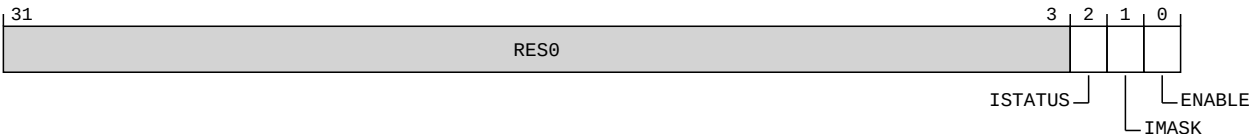
It is IMPLEMENTATION DEFINED whether CNTP_CTL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘[Power and reset domains for the system level implementation of the Generic Timer](#)’.

Attributes

CNTP_CTL is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0, then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTP_TVAL](#) continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CTL

CNTP_CTL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

‘[CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames](#)’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CTL is accessible in that frame if the value of [CNTACR<n>.RWPT](#) is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_CTL is accessible in that frame if both:
 - CNTP_CTL is accessible in the corresponding CNTBaseN frame:
 - The value of [CNTEL0ACR.ELOPTEN](#) is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

CNTP_CTL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x02C	CNTP_CTL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTELOBaseN	0x02C	CNTP_CTL

Accesses to this register are RW.

F4.5.11 CNTP_CVAL, Counter-timer Physical Timer CompareValue

The CNTP_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the EL1 physical timer.

Configuration

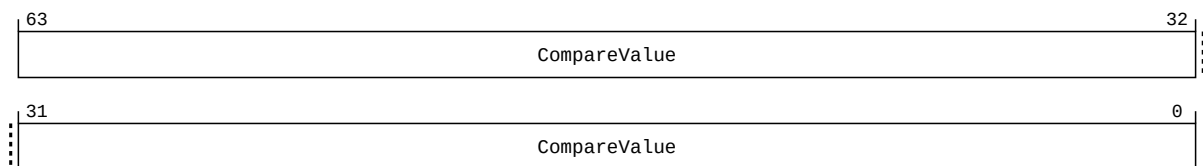
It is IMPLEMENTATION DEFINED whether CNTP_CVAL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘[Power and reset domains for the system level implementation of the Generic Timer](#)’.

Attributes

CNTP_CVAL is a 64-bit register.

Field descriptions



CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When [CNTP_CTL.ENABLE](#) is 1, the timer condition is met when ([CNTPCT](#) - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- [CNTP_CTL.ISTATUS](#) is set to 1.
- An interrupt is generated if [CNTP_CTL.IMASK](#) is 0.

When [CNTP_CTL.ENABLE](#) is 0, the timer condition is not met, but [CNTPCT](#) continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL

CNTP_CVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

‘[CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames](#)’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CVAL is accessible in that frame if the value of [CNTACR<n>.RWPT](#) is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_CVAL is accessible in that frame if both:
 - CNTP_CVAL is accessible in the corresponding CNTBaseN frame:
 - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTP_CVAL register must be accessible as an atomic 64-bit value.

CNTP_CVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x020	CNTP_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x024	CNTP_CVAL	63:32

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x020	CNTP_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x024	CNTP_CVAL	63:32

Accesses to this register are RW.

F4.5.12 CNTP_TVAL, Counter-timer Physical Timer TimerValue

The CNTP_TVAL characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configuration

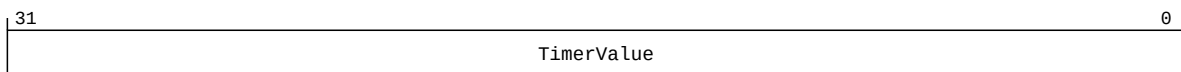
It is IMPLEMENTATION DEFINED whether CNTP_TVAL is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

Attributes

CNTP_TVAL is a 32-bit register.

Field descriptions



TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If [CNTP_CTL.ENABLE](#) is 0, the value returned is UNKNOWN.
- If [CNTP_CTL.ENABLE](#) is 1, the value returned is (CompareValue - [CNTPCT](#)).

On a write of this register, CompareValue is set to ([CNTPCT](#) + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When [CNTP_CTL.ENABLE](#) is 1, the timer condition is met when ([CNTPCT](#) - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- [CNTP_CTL.ISTATUS](#) is set to 1.
- If [CNTP_CTL.IMASK](#) is 0, an interrupt is generated.

When [CNTP_CTL.ENABLE](#) is 0, the timer condition is not met, but [CNTPCT](#) continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_TVAL

CNTP_TVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

[‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’](#) describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_TVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_TVAL is accessible in that frame if both:
 - CNTP_TVAL is accessible in the corresponding CNTBaseN frame:
 - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

CNTP_TVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x028	CNTP_TVAL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x028	CNTP_TVAL

Accesses to this register are RW.

F4.5.13 CNTSR, Counter Status Register

The CNTSR characteristics are:

Purpose

Provides counter resolution status information.

Configuration

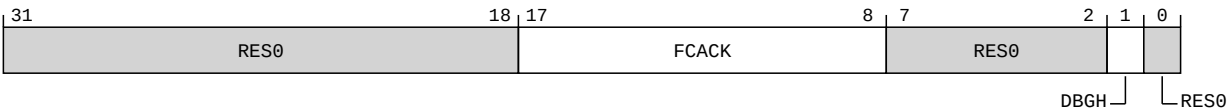
It is IMPLEMENTATION DEFINED whether CNTSR is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTSR is a 32-bit register.

Field descriptions



Bits [31:18]

Reserved, RES0.

FCACK, bits [17:8]

Frequency Change Acknowledge. Indicates the currently selected entry in the Frequency modes table, see ‘The Frequency modes table’.

The reset behavior of this field is:

- On a Timer reset, this field resets to the expression 0x000000.

Bits [7:2]

Reserved, RES0.

DBGH, bit [1]

Indicates whether the counter is halted because the Halt-on-debug signal is asserted:

DBGH	Meaning
0b0	Counter is not halted.
0b1	Counter is halted.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing CNTSR

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTSR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	0x004	CNTSR

Accesses to this register are RO.

F4.5.14 CNTTIDR, Counter-timer Timer ID Register

The CNTTIDR characteristics are:

Purpose

Indicates the implemented timers in the physical address space, and their features. For each value of N from 0 to 7 it indicates whether:

- Frame CNTBaseN is a view of an implemented timer.
- Frame CNTBaseN has a second view, CNTEL0BaseN.
- Frame CNTBaseN has a virtual timer capability.

Configuration

It is IMPLEMENTATION DEFINED whether CNTTIDR is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

Attributes

CNTTIDR is a 32-bit register.

Field descriptions

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
Frame7	Frame6	Frame5	Frame4	Frame3	Frame2	Frame1	Frame0								

Frame<n>, bits [4n+3:4n], for n = 7 to 0

A 4-bit field indicating the features of frame CNTBase<n>.

Bit[3] of the field is RES0.

Bit[2], the FEL0 subfield, indicates whether frame CNTBase<n> has a second view, CNTEL0Base<n>. The possible values of this bit are:

Bit[2]	Meaning
0b0	Frame<n> does not have a second view. The CNTEL0ACR register in the first view of the frame is RES0.
0b1	Frame<n> has a second view, CNTEL0Base<n>.

If bit[0] is 0, bit[2] is RES0.

Bit[1], the FVI subfield, indicates whether both:

- Frame CNTBase<n> implements the virtual timer registers [CNTV_CVAL](#), [CNTV_TVAL](#), and [CNTV_CTL](#).
- This CNTCTLBase frame implements the virtual timer offset register [CNTVOFF<n>](#).

The possible values of bit[1] are:

Bit[1]	Meaning
0b0	Frame<n> does not have virtual capability. The virtual time and offset registers are RES0.
0b1	Frame<n> has virtual capability. The virtual time and offset registers are implemented.

If bit[0] is 0, bit[1] is RES0.

Bit[0], the FI subfield, indicates whether frame CNTBase<n> is implemented. The possible values of this bit are:

Bit[0]	Meaning
0b0	Frame<n> is not implemented. All registers associated with the frame are RES0.
0b1	Frame<n> is implemented.

Accessing CNTTIDR

In a system that recognizes two Security states, this register is accessible by both Secure and Non-secure accesses.

CNTTIDR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	0x008	CNTTIDR

Accesses to this register are RO.

F4.5.15 CNTVCT, Counter-timer Virtual Count

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value.

Configuration

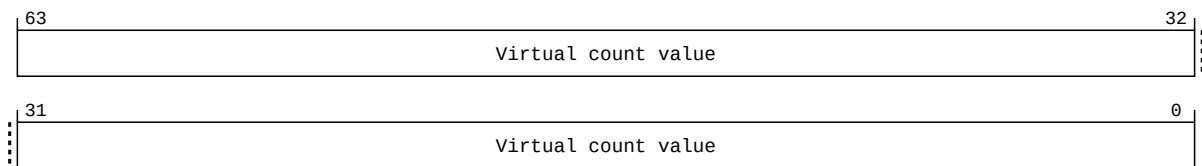
It is IMPLEMENTATION DEFINED whether CNTVCT is implemented in the Core power domain or in the Debug power domain

For more information, see ‘[Power and reset domains for the system level implementation of the Generic Timer](#)’.

Attributes

CNTVCT is a 64-bit register.

Field descriptions



VirtualCount, bits [63:0]

Virtual count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCT

CNTVCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

‘[CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames](#)’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTVCT is accessible in that frame, as a RO register, if the value of [CNTACR<n>.RVCT](#) is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTVCT is accessible in that frame if both:
 - CNTVCT is accessible in the corresponding CNTBaseN frame.
 - The value of [CNTEL0ACR.EL0VCTEN](#) is 1.

- Otherwise, the CNTVCT address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTVCT register must be accessible as an atomic 64-bit value.

CNTVCT can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x008	CNTVCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x00C	CNTVCT	63:32

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x008	CNTVCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x00C	CNTVCT	63:32

Accesses to this register are RO.

F4.5.16 CNTVOFF, Counter-timer Virtual Offset

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset for a CNTBaseN frame that has virtual timer capability. This is the offset between real time and virtual time.

Configuration

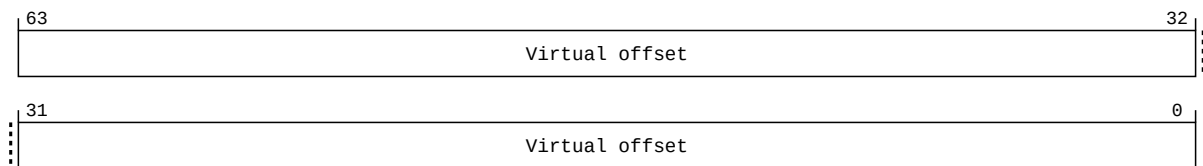
It is IMPLEMENTATION DEFINED whether CNTVOFF is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

Attributes

CNTVOFF is a 64-bit register.

Field descriptions



VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF

CNTVOFF is implemented, as a RO register, in any implemented CNTBaseN frame that has virtual timer capability.

[‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’](#) describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTVOFF is accessible in that frame, as a RO register, if the value of [CNTACR<n>.RVOFF](#) is 1.
- Otherwise, the CNTVOFF address in that frame is RAZ/WI.

Note

CNTVOFF is never visible in any CNTEL0BaseN frame. This means that the CNTVOFF address in any implemented CNTEL0BaseN frame is RAZ/WI.

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF register must be accessible as an atomic 64-bit value.

CNTVOFF can be accessed through the memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTBaseN	0x018	31:0

Accesses to this register are RO.

Component	Frame	Offset	Range
Timer	CNTBaseN	0x01C	63:32

Accesses to this register are RO.

F4.5.17 CNTVOFF<n>, Counter-timer Virtual Offsets, n = 0 - 7

The CNTVOFF<n> characteristics are:

Purpose

Holds the 64-bit virtual offset for frame CNTBase<n>. This is the offset between real time and virtual time.

Configuration

It is IMPLEMENTATION DEFINED whether CNTVOFF<n> is implemented in the Core power domain or in the Debug power domain

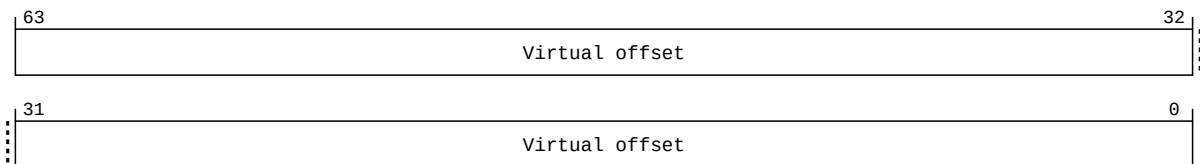
For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

Implementation of this register is OPTIONAL.

Attributes

CNTVOFF<n> is a 64-bit register.

Field descriptions



VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF<n>

In the CNTCTBase frame a CNTVOFF<n> register must be implemented, as a RW register, for each CNTBaseN frame that has virtual timer capability. For more information, see [‘CNTCTBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’](#).

Note

The value of <n> in an instance of CNTVOFF<n> specifies the value of N for the associated CNTBaseN frame.

In a system that recognizes two Security states, for any CNTVOFF<n> register in the CNTCTBase frame:

- CNTVOFF<n> is always accessible by Secure accesses.
- [CNTNSAR.NS<n>](#) determines whether CNTVOFF<n> is accessible by Non-secure accesses.

The register location of any unimplemented CNTVOFF<n> register in the CNTCTBase frame is RAZ/WI.

The CNTVOFF<n> register is accessible in the CNTBaseN frame using [CNTVOFF](#).

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF<n> registers must be accessible as atomic 64-bit values.

CNTVOFF<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTCTLBase	$0x080 + (8 * n)$	31:0

Accesses to this register are RW.

Component	Frame	Offset	Range
Timer	CNTCTLBase	$0x084 + (8 * n)$	63:32

Accesses to this register are RW.

F4.5.18 CNTV_CTL, Counter-timer Virtual Timer Control

The CNTV_CTL characteristics are:

Purpose

Control register for the virtual timer.

Configuration

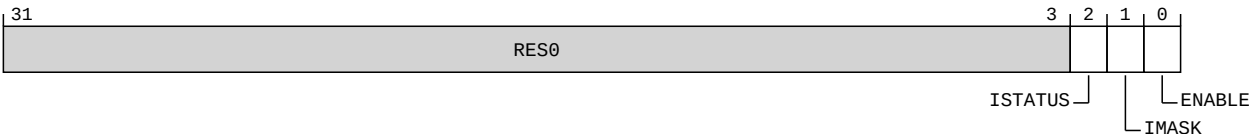
It is IMPLEMENTATION DEFINED whether CNTV_CTL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTV_CTL is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0, then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from [CNTV_TVAL](#) continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CTL

CNTV_CTL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

‘[CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames](#)’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CTL is accessible in that frame if the value of [CNTACR<n>.RWVT](#) is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_CTL is accessible in that frame if both:
 - CNTV_CTL is accessible in the corresponding CNTBaseN frame:
 - The value of [CNTEL0ACR.EL0VTEN](#) is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

CNTV_CTL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x03C	CNTV_CTL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x03C	CNTV_CTL

Accesses to this register are RW.

F4.5.19 CNTV_CVAL, Counter-timer Virtual Timer CompareValue

The CNTV_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the virtual timer.

Configuration

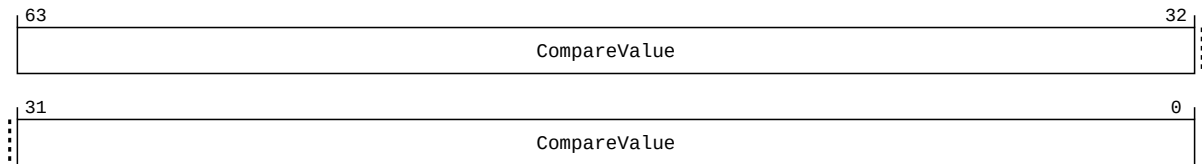
It is IMPLEMENTATION DEFINED whether CNTV_CVAL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘[Power and reset domains for the system level implementation of the Generic Timer](#)’.

Attributes

CNTV_CVAL is a 64-bit register.

Field descriptions



CompareValue, bits [63:0]

Holds the virtual timer CompareValue.

When [CNTV_CTL.ENABLE](#) is 1, the timer condition is met when ([CNTVCT](#) - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- [CNTV_CTL.ISTATUS](#) is set to 1.
- An interrupt is generated if [CNTV_CTL.IMASK](#) is 0.

When [CNTV_CTL.ENABLE](#) is 0, the timer condition is not met, but [CNTVCT](#) continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CVAL

CNTV_CVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

‘[CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames](#)’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CVAL is accessible in that frame if the value of [CNTACR<n>.RWVT](#) is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_CVAL is accessible in that frame if both:
 - CNTV_CVAL is accessible in the corresponding CNTBaseN frame:
 - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTV_CVAL register must be accessible as an atomic 64-bit value.

CNTV_CVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x030	CNTV_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x034	CNTV_CVAL	63:32

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x030	CNTV_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x034	CNTV_CVAL	63:32

Accesses to this register are RW.

F4.5.20 CNTV_TVAL, Counter-timer Virtual Timer TimerValue

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configuration

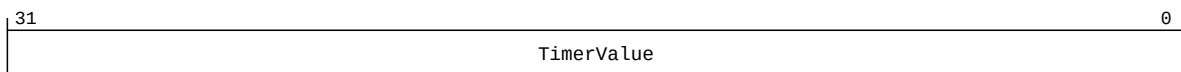
It is IMPLEMENTATION DEFINED whether CNTV_TVAL is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions



TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If [CNTV_CTL.ENABLE](#) is 0, the value returned is UNKNOWN.
- If [CNTV_CTL.ENABLE](#) is 1, the value returned is (CompareValue - [CNTVCT](#)).

On a write of this register, CompareValue is set to ([CNTVCT](#) + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When [CNTV_CTL.ENABLE](#) is 1, the timer condition is met when ([CNTVCT](#) - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- [CNTV_CTL.ISTATUS](#) is set to 1.
- If [CNTV_CTL.IMASK](#) is 0, an interrupt is generated.

When [CNTV_CTL.ENABLE](#) is 0, the timer condition is not met, but [CNTVCT](#) continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_TVAL

CNTV_TVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

[‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’](#) describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_TVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_TVAL is accessible in that frame if both:
 - CNTV_TVAL is accessible in the corresponding CNTBaseN frame.
 - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

CNTV_TVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x038	CNTV_TVAL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x038	CNTV_TVAL

Accesses to this register are RW.

F4.5.21 CounterID<n>, Counter ID registers, n = 0 - 11

The CounterID<n> characteristics are:

Purpose

IMPLEMENTATION DEFINED identification registers 0 to 11 for the memory-mapped Generic Timer.

Configuration

It is IMPLEMENTATION DEFINED whether CounterID<n> is implemented in the Core power domain or in the Debug power domain

For more information, see [‘Power and reset domains for the system level implementation of the Generic Timer’](#).

These registers are implemented independently in each of the implemented Generic Timer memory-mapped frames.

If the implementation of the Counter ID registers requires an architecture version, the value for this version of the Arm Generic Timer is version 0.

The Counter ID registers can be implemented as a set of CoreSight ID registers, comprising Peripheral ID Registers and Component ID Registers. An implementation of these registers for the Generic Timer must use a Component class value of 0xF.

Attributes

CounterID<n> is a 32-bit register.

Field descriptions

31	IMPLEMENTATION DEFINED	0
----	------------------------	---

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing CounterID<n>

These registers must be implemented, as RO registers, in every implemented Generic Timer memory-mapped frame.

For the CNTCTLBase frame, in a system that recognizes two Security states these registers are accessible by both Secure and Non-secure accesses.

For the CNTControlBase frame, in a system that supports Secure and Non-secure physical address spaces, the frame is implemented only in the Secure physical address space, meaning these registers are implemented only in the Secure physical address space.

For the CNTReadBase frame, these registers are accessible in all physical address spaces.

For the CNTBaseN frames, [‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’](#) describes the status fields that identify whether a frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

CounterID<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	$0 \times \text{FD0} + (4 * n)$	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTReadBase	$0 \times \text{FD0} + (4 * n)$	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTBaseN	$0 \times \text{FD0} + (4 * n)$	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	$0 \times \text{FD0} + (4 * n)$	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTCTLBase	$0 \times \text{FD0} + (4 * n)$	CounterID<n>

Accesses to this register are RO.

Part G
Architectural Pseudocode

Chapter G1

R-profile Architecture Pseudocode

This chapter contains pseudocode that describes many features of the R-profile architecture. It contains the following sections:

- [Pseudocode for AArch32 operation](#)
- [Shared pseudocode](#)

G1.1 Pseudocode for AArch32 operation

This section holds the pseudocode for execution in AArch32 state. Functions that are listed in this section are identified as `AArch32.FunctionName` or `FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch32/debug/BreakpointMatch`.

Note

Many AArch32 pseudocode functions have not been updated to show the constraints on the Armv7 UNPREDICTABLE behaviors that are described in [Architectural Constraints on UNPREDICTABLE Behaviors](#). Where AArch32 pseudocode shows something to be UNPREDICTABLE, check [Architectural Constraints on UNPREDICTABLE Behaviors](#) for possible constraints on the permitted behavior.

The top-level sections of the AArch32 pseudocode hierarchy are:

- [aarch32/debug](#).
- [aarch32/exceptions](#).
- [aarch32/functions](#).
- [aarch32/translation](#).

G1.1.1 aarch32/debug

This section includes the following pseudocode functions:

- [AArch32_VCRMatch](#)
- [AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled](#)
- [ExternalHypInvasiveDebugEnabled](#)
- [ExternalHypNoninvasiveDebugEnabled](#)
- [AArch32_BreakpointMatch](#)
- [AArch32_BreakpointValueMatch](#)
- [AArch32_ReservedBreakpointType](#)
- [AArch32_StateMatch](#)
- [AArch32_GenerateDebugExceptions](#)
- [AArch32_GenerateDebugExceptionsFrom](#)
- [AArch32_IncrementCycleCounter](#)
- [AArch32_IncrementEventCounter](#)
- [AArch32_PMUCycle](#)
- [AArch32_EnterHypModeInDebugState](#)
- [AArch32_EnterModeInDebugState](#)
- [AArch32_WatchpointByteMatch](#)
- [AArch32_WatchpointMatch](#)

G1.1.1.1 AArch32_VCRMatch

```
// AArch32_VCRMatch()
// =====

func AArch32_VCRMatch(vaddress : bits(32)) => boolean
begin

    var match : boolean;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        var match_word : bits(32) = Zeros{};

        let ss : SecurityState = CurrentSecurityState();
```

```

    if vaddress[31:5] == ExcVectorBase() [31:5] then
        if HaveEL(EL3) && ss == SS_NonSecure then
            match_word[UInt(vaddress[4:2]) + 24] = '1'; // Non-secure vectors
        else
            match_word[UInt(vaddress[4:2]) + 0] = '1'; // Secure vectors (or no EL3)
        end;
    end;
    // Mask out bits not corresponding to vectors.
    var mask : bits(32);
    if !HaveEL(EL3) then
        mask = '00000000'::'00000000'::'00000000'::'11011110'; // DBGVCR[31:8] are RES0
    elsif !ELUsingAArch32(EL3) then
        mask = '11011110'::'00000000'::'00000000'::'11011110'; // DBGVCR[15:8] are RES0
    else
        mask = '11011110'::'00000000'::'11011100'::'11011110';
    end;

    match_word = match_word AND DBGVCR() AND mask;
    match = !IsZero(match_word);

    // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
    if !IsZero(match_word[28:27,12:11,4:3]) && DebugTarget() == PSTATE.EL then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);
    end;

    if !IsZero(vaddress[1:0]) && match then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
    end;
else
    match = FALSE;
end;

return match;
end;

```

G1.1.1.2 AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled

```

// AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

func AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled() => boolean
begin
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32_SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE; end;
    return DBGEN == HIGH && SPIDEN == HIGH;
end;

```

G1.1.1.3 ExternalHypInvasiveDebugEnabled

```

// ExternalHypInvasiveDebugEnabled()
// =====

// In the recommended interface, ExternalHypInvasiveDebugEnabled returns the state
// of the (DBGEN AND HIDDEN) signal.

func ExternalHypInvasiveDebugEnabled() => boolean
begin
    return ExternalInvasiveDebugEnabled() && HIDDEN == HIGH;
end;

```

G1.1.1.4 ExternalHypNoninvasiveDebugEnabled

```
// ExternalHypNoninvasiveDebugEnabled()
// =====
// In the recommended interface, ExternalHypNoninvasiveDebugEnabled returns the
// state of the (DBGGEN OR NIDEN) AND (HIDEN OR HNIDEN) signal.

func ExternalHypNoninvasiveDebugEnabled() => boolean
begin
    return ExternalNoninvasiveDebugEnabled() && (HIDEN == HIGH || HNIDEN == HIGH);
end;
```

G1.1.1.5 AArch32_BreakpointMatch

```
// AArch32_BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

func AArch32_BreakpointMatch(n : integer, vaddress : bits(32), accdesc : AccessDescriptor,
                             size : integer) => BreakpointInfo
begin
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();

    var brkptinfo : BreakpointInfo;
    let enabled : boolean = DBGBCR(n).E == '1';
    let isbreakpnt : boolean = TRUE;
    let linked : boolean = DBGBCR(n).BT == '0x01';
    let linked_to : boolean = FALSE;
    let linked_n : integer{} = UInt(DBGBCR(n).LBN);

    let state_match : boolean = AArch32_StateMatch(DBGBCR(n).SSC, DBGBCR(n).HMC, DBGBCR(n).PMC,
                                                    linked, linked_n, isbreakpnt, accdesc);
    var (value_match, value_mismatch) = AArch32_BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether this breakpoint generates a Breakpoint debug
        // event.
        let (match_i, mismatch_i) = AArch32_BreakpointValueMatch(n, vaddress + 2, linked_to);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        end;

        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);
        end;
    end;

    if vaddress[1] == '1' && DBGBCR(n).BAS == '1111' then
        // The above notwithstanding, if DBGBCR(n).BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBCR(n)+2.
        if value_match then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        end;

        if !value_mismatch then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);
        end;
    end;

    brkptinfo.match = value_match && state_match && enabled;
    brkptinfo.mismatch = value_mismatch && state_match && enabled;
```

```
    return brkptinfo;
end;
```

G1.1.1.6 AArch32_BreakpointValueMatch

```
// AArch32_BreakpointValueMatch()
// =====
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

func AArch32_BreakpointValueMatch(n_in : integer, vaddress : bits(32),
                                  linked_to : boolean) => (boolean, boolean)
begin

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    // matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    var n : integer = n_in;
    var c : Constraint;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                              Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE, FALSE); end;
    end;

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if DBGBCR(n).E == '0' then return (FALSE, FALSE); end;

    var dbgtype : bits(4) = DBGBCR(n).BT;

    (c, dbgtype) = AArch32_ReservedBreakpointType(n, dbgtype);
    if c == Constraint_DISABLED then return (FALSE, FALSE); end;
    // Otherwise the dbgtype value returned by AArch32_ReservedBreakpointType is valid.

    // Determine what to compare against.
    let match_addr : boolean    = (dbgtype == '0x0x');
    let mismatch : boolean     = (dbgtype == '010x');
    let match_vmid : boolean    = (dbgtype == '10xx');
    let match_cid : boolean     = (dbgtype == 'x01x');
    let linking_enabled : boolean = (dbgtype == 'xxx1');

    // If called from StateMatch, is CONSTRAINED UNPREDICTABLE if the
    // breakpoint is not programmed with linking enabled.
    if linked_to && !linking_enabled then
        if !ConstrainUnpredictableBool(Unpredictable_BPLINKINGDISABLED) then
            return (FALSE, FALSE);
        end;
    end;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linking_enabled && !match_addr then
        return (FALSE, FALSE);
    end;

    var bvr_match : boolean = FALSE;
    var bxvr_match : boolean = FALSE;

    // Do the comparison.
```



```

if match_addr then
    let byte : integer = UInt(vaddress[1:0]);
    assert byte IN {0,2}; // "vaddress" is halfword aligned

    let byte_select_match : boolean = (DBGBCR(n).BAS[byte] == '1');
    bvr_match = (vaddress[31:2] == DBGGBVR(n)[31:2]) && byte_select_match;

elseif match_cid then
    bvr_match = (PSTATE.EL != EL2 && CONTEXTIDR() == DBGGBVR(n)[31:0]);
end;

if match_vmid then
    let vmid : bits(NUM_VMIDBITS) = VMID();
    var bvr_vmid : bits(NUM_VMIDBITS);

    if ELUsingAArch32(EL2) then
        bvr_vmid = ZeroExtend{NUM_VMIDBITS}(DBGGBXVR(n)[7:0]);
    end;

    bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == bvr_vmid);

end;
let bvr_match_valid : boolean = (match_addr || match_cid);
let bxvr_match_valid : boolean = match_vmid;

let match : boolean = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

return (match && !mismatch, !match && mismatch);
end;

```

G1.1.1.7 AArch32_ReservedBreakpointType

```

// AArch32_ReservedBreakpointType()
// =====
// Checks if the given DBGBCR(n).BT value is reserved and will generate Constrained Unpredictable
// behavior, otherwise returns Constraint_NONE.

func AArch32_ReservedBreakpointType(n : integer, bt_in : bits(4)) => (Constraint, bits(4))
begin
    var bt : bits(4) = bt_in;
    var reserved : boolean = FALSE;
    let context_aware : boolean = IsContextAwareBreakpoint(n);

    // Address mismatch
    if bt == '010x' && HaltOnBreakpointOrWatchpoint() then
        reserved = TRUE;
    end;

    // Context matching
    if bt != '0x0x' && !context_aware then
        reserved = TRUE;
    end;

    // EL2 extension
    if bt == '1xxx' && !HaveEL(EL2) then
        reserved = TRUE;
    end;

    // Reserved
    if bt IN {'011x', '11xx'} then
        reserved = TRUE;
    end;

    if reserved then
        var c : Constraint;
        (c, bt) = ConstrainUnpredictableBits{4}(Unpredictable_RESBPTYPE);
    end;
end;

```

```

    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then
        return (c, ARBITRARY : bits(4));
    // Otherwise the value returned by ConstrainsUnpredictableBits must be a not-reserved value
    end;
end;

return (Constraint_NONE, bt);
end;

```

G1.1.1.8 AArch32_StateMatch

```

// AArch32_StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

func AArch32_StateMatch(ssc_in : bits(2), hmc_in : bit, pxc_in : bits(2), linked_in : boolean,
    linked_n_in : integer, isbreakpnt : boolean,
    accdesc : AccessDescriptor) => boolean
begin
    // "ssc_in","hmc_in","pxc_in" are the control fields from the DBGBCR(n) or DBGWCR(n) register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR(n) or DBGWCR(n) register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "accdesc" describes the properties of the access being matched.
    var hmc : bit = hmc_in;
    var ssc : bits(2) = ssc_in;
    var pxc : bits(2) = pxc_in;
    var linked : boolean = linked_in;
    var linked_n : integer = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    var c : Constraint;
    (c, ssc, hmc, pxc) = CheckValidStateMatch(ssc, hmc, pxc, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE; end;
    // Otherwise the hmc,ssc,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    let pl2_match : boolean = HaveEL(EL2) && ((hmc == '1' && (ssc::pxc != '1000')) || ssc == '11');
    let pl1_match : boolean = pxc[0] == '1';
    let pl0_match : boolean = pxc[1] == '1';
    let ssu_match : boolean = isbreakpnt && hmc == '0' && pxc == '00' && ssc != '11';

    var priv_match : boolean;
    if ssu_match then
        priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
    else
        case accdesc.el of
            when EL3 => priv_match = pl1_match; // EL3 and EL1 are both PL1
            when EL2 => priv_match = pl2_match;
            when EL1 => priv_match = pl1_match;
            when EL0 => priv_match = pl0_match;
        end;
    end;

    // Security state match
    var ss_match : boolean;
    case ssc of
        when '00' => ss_match = TRUE; // Both
        when '01' => ss_match = accdesc.ss == SS_NonSecure; // Non-secure only
        when '10' => ss_match = accdesc.ss == SS_Secure; // Secure only
        when '11' => ss_match = (hmc == '1' || accdesc.ss == SS_Secure); // HMC=1 -> Both,
        // HMC=0 -> Secure only
    end;
end;

```

```

var linked_match : boolean = FALSE;

if linked then
    // "linked_n" must be an enabled context-aware breakpoint unit.
    // If it is not context-aware then it is CONSTRAINED UNPREDICTABLE whether
    // this gives no match, gives a match without linking, or linked_n is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    if !IsContextAwareBreakpoint(linked_n) then
        let (first_ctx_cmp, last_ctx_cmp) : (integer, integer) = ContextAwareBreakpointRange();
        (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
                                                    Unpredictable_BFNOTCTXCMP);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};

        case c of
            when Constraint_DISABLED => return FALSE;      // Disabled
            when Constraint_NONE =>    linked = FALSE;      // No linking
            // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
        end;
    end;

    let vaddress : bits(32) = ARBITRARY : bits(32);
    let linked_to : boolean = TRUE;
    (linked_match, -) = AArch32_BreakpointValueMatch(linked_n, vaddress, linked_to);
end;

return priv_match && ss_match && (!linked || linked_match);
end;

```

G1.1.1.9 AArch32_GenerateDebugExceptions

```

// AArch32_GenerateDebugExceptions()
// =====

func AArch32_GenerateDebugExceptions() => boolean
begin
    let ss : SecurityState = CurrentSecurityState();
    return AArch32_GenerateDebugExceptionsFrom(PSTATE.EL, ss);
end;

```

G1.1.1.10 AArch32_GenerateDebugExceptionsFrom

```

// AArch32_GenerateDebugExceptionsFrom()
// =====

func AArch32_GenerateDebugExceptionsFrom(from_el : bits(2), from_state : SecurityState) => boolean
begin
    if DBGOSLSR().OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    end;

    var enabled : boolean;
    enabled = from_el != EL2;

    return enabled;
end;

```

G1.1.1.11 AArch32_IncrementCycleCounter

```

// AArch32_IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

func AArch32_IncrementCycleCounter()
begin

```

```

    if !CountPMUEvents(CYCLE_COUNTER_ID) then return; end;
    var d : bit = PMCR().D;    // Check divide-by-64
    var lc : bit = PMCR().LC;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0'; end;
    if d == '1' && !HasElapsed64Cycles() then return; end;

    let old_value : integer = UInt(PMCCNTR());
    let new_value : integer = old_value + 1;
    PMCCNTR() = new_value[63:0];

    let ovflw : integer{} = if lc == '1' then 64 else 32;

    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET().C = '1';
    end;

    return;
end;

```

G1.1.1.12 AArch32_IncrementEventCounter

```

// AArch32_IncrementEventCounter()
// =====
// Increment the specified event counter 'idx' by the specified amount 'increment'.
func AArch32_IncrementEventCounter(idx : integer, increment : integer)
begin
    var old_value : integer;
    var new_value : integer;

    old_value = UInt(PMEVCNTR(idx));
    new_value = old_value + increment;

    PMEVCNTR(idx) = new_value[31:0];
    let ovflw : integer{} = 32;
    if old_value[64:ovflw] != new_value[64:ovflw] then
        PMOVSSET()[idx] = '1';
        // Check for the CHAIN event from an even counter
        if (idx[0] == '0' && idx + 1 < NUM_PMU_COUNTERS &&
            (GetPMUCounterRange(idx) == GetPMUCounterRange(idx+1) ||
             ConstrainUnpredictableBool(Unpredictable_COUNT_CHAIN))) then
            // If PMU counters idx and idx+1 are not in same range,
            // it is CONSTRAINED UNPREDICTABLE if CHAIN event is counted
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
        end;
    end;

    return;
end;

```

G1.1.1.13 AArch32_PMUCycle

```

// AArch32_PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

func AArch32_PMUCycle()
begin
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    end;
    if (!IsWFIsleep && !IsWFEsleep) || ImpDefBool("PMU counts cycles in WFI or WFE state") then
        PMUEvent(PMU_EVENT_CPU_CYCLES);
    end;
end;

```

```

end;
let counters : integer = NUM_PMU_COUNTERS;
if counters != 0 then
    for idx = 0 to counters - 1 do
        if CountPMUEvents(idx) then
            let accumulated : integer = PMUEventAccumulator[[idx]];
            AArch32_IncrementEventCounter(idx, accumulated);
        end;
        PMUEventAccumulator[[idx]] = 0;
    end;
end;
AArch32_IncrementCycleCounter();
CheckForPMUOverflow();
end;

```

G1.1.1.14 AArch32_EnterHypModeInDebugState

```

// AArch32_EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

noreturn func AArch32_EnterHypModeInDebugState(except : ExceptionRecord)
begin
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    AArch32_ReportHypEntry(except);
    AArch32_WriteMode(M32_Hyp);
    SPSR_curr() = ARBITRARY : bits(32);
    ELR_hyp() = ARBITRARY : bits(32);
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.[SS,A,I,F] are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.[SS,A,I,F] = ARBITRARY : bits(4);
    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);
    PSTATE.E = HSCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    EDSCR().ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction(EndOfInstructionReason_Exception);
end;

```

G1.1.1.15 AArch32_EnterModeInDebugState

```

// AArch32_EnterModeInDebugState()
// =====
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

noreturn func AArch32_EnterModeInDebugState(target_mode : bits(5))
begin
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    AArch32_WriteMode(target_mode);
    SPSR_curr() = ARBITRARY : bits(32);
    R(14) = ARBITRARY : bits(32);
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.[SS,A,I,F] are not observable so behave as UNKNOWN.
    PSTATE.T = '1'; // PSTATE.J is RES0
    PSTATE.[SS,A,I,F] = ARBITRARY : bits(4);
    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);

```

```

PSTATE.E = SCTL.R().EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
EDSCR().ERR = '1';
UpdateEDSCRFields(); // Update EDSCR PE state flags.

EndOfInstruction(EndOfInstructionReason_Exception);
end;

```

G1.1.1.16 AArch32_WatchpointByteMatch

```

// AArch32_WatchpointByteMatch()
// =====

func AArch32_WatchpointByteMatch(n : integer, vaddress : bits(32)) => boolean
begin
    let dbgtop : integer{} = 31;
    let cmpbottom : integer{} = if DBGWVR(n)[2] == '1' then 2 else 3; // Word or doubleword
    var bottom = cmpbottom;
    let select : integer{} = UInt(vaddress[cmpbottom-1:0]);
    var byte_select_match : boolean = (DBGWCR(n).BAS[select] != '0');
    var mask : integer{} = UInt(DBGWCR(n).MASK);

    // If DBGWCR(n).MASK is a nonzero value and DBGWCR(n).BAS is not set to '11111111', or
    // DBGWCR(n).BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR(n).BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        let LSB : bits(8) = (DBGWCR(n).BAS AND NOT(DBGWCR(n).BAS - 1));
        let MSB : bits(8) = (DBGWCR(n).BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3; // For the whole doubleword
        end;
    end;

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        var c : Constraint;
        var unpred_mask : integer;
        (c, unpred_mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED => return FALSE; // Disabled
            when Constraint_NONE => mask = 0; // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value
            otherwise => mask = unpred_mask as integer{3..31};
        end;
    end;

    let cmpmsb : integer{} = dbgtop;
    let cmplsmb : integer{} = if mask > bottom then mask else bottom;
    let bottombit : integer{} = bottom;
    var WVR_match : boolean = (vaddress[cmpmsb:cmplsmb] == DBGWVR(n)[cmpmsb:cmplsmb]);
    if mask > bottom then
        // If masked bits of DBGWVR(n) are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR(n)[cmplsmb-1:bottombit]) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
        end;
    end;

    return (WVR_match && byte_select_match);
end;

```

G1.1.1.17 AArch32_WatchpointMatch

```
// AArch32_WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

func AArch32_WatchpointMatch(n : integer, vaddress : bits(32), size : integer,
                             accdesc : AccessDescriptor) => WatchpointInfo
begin
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    let enabled : boolean      = DBGWCR(n).E == '1';
    let linked : boolean      = DBGWCR(n).WT == '1';
    let isbreakpnt : boolean  = FALSE;
    let linked_n : integer{} = UInt(DBGWCR_EL1(n).LBN);
    let state_match : boolean = AArch32_StateMatch(DBGWCR(n).SSC, DBGWCR(n).HMC, DBGWCR(n).PAC,
                                                    linked, linked_n, isbreakpnt, accdesc);

    var watchptinfo : WatchpointInfo;
    var ls_match : boolean;
    case DBGWCR(n).LSC[1:0] of
        when '00' => ls_match = FALSE;
        when '01' => ls_match = accdesc.read;
        when '10' => ls_match = accdesc.write || accdesc.acctype == AccessType_DC;
        when '11' => ls_match = TRUE;
    end;

    var value_match : boolean = FALSE;
    watchptinfo.vaddress = ZeroExtend{64}(vaddress);
    for byte = 0 to size - 1 do
        if (!value_match &&
            !AddressInNaturallyAlignedBlock(watchptinfo.vaddress,
                                             ZeroExtend{64}(vaddress + byte))) then
            // Watchpoint should report an address which is in
            // the naturally aligned block of the matched address.
            watchptinfo.vaddress = ZeroExtend{64}(vaddress + byte);
        end;
        value_match = value_match || AArch32_WatchpointByteMatch(n, vaddress + byte);
    end;

    watchptinfo.value_match = value_match && state_match && ls_match && enabled;
    return watchptinfo;
end;
```

G1.1.2 aarch32/exceptions

This section includes the following pseudocode functions:

- [AArch32_Abort](#)
- [AArch32_AbortSyndrome](#)
- [AArch32_CheckPCAlignment](#)
- [AArch32_CommonFaultStatus](#)
- [AArch32_ReportDataAbort](#)
- [AArch32_ReportPrefetchAbort](#)
- [AArch32_TakeDataAbortException](#)
- [AArch32_TakePrefetchAbortException](#)
- [AArch32_TakePhysicalFIQException](#)
- [AArch32_TakePhysicalIRQException](#)
- [AArch32_TakePhysicalSErrorException](#)

- [AArch32_TakeVirtualFIQException](#)
- [AArch32_TakeVirtualIRQException](#)
- [AArch32_TakeVirtualSErrorException](#)
- [AArch32_SoftwareBreakpoint](#)
- [DebugException](#)
- [AArch32_CheckAdvSIMDOrFPRegisterTraps](#)
- [AArch32_ExceptionClass](#)
- [AArch32_ReportHypEntry](#)
- [AArch32_ResetControlRegisters](#)
- [AArch32_TakeReset](#)
- [ExcVectorBase](#)
- [AArch32_FPTrappedException](#)
- [AArch32_CallHypervisor](#)
- [AArch32_CallSupervisor](#)
- [AArch32_TakeHVCEException](#)
- [AArch32_TakeSVCException](#)
- [AArch32_EnterHypMode](#)
- [AArch32_EnterMode](#)
- [AArch32_CheckAdvSIMDOrFPEEnabled](#)
- [AArch32_CheckFPAdvSIMDTrap](#)
- [AArch32_CheckForSMCUndefOrTrap](#)
- [AArch32_CheckForWfxTrap](#)
- [AArch32_CheckITEnabled](#)
- [AArch32_CheckIllegalState](#)
- [AArch32_CheckSETENDEnabled](#)
- [AArch32_SystemAccessTrap](#)
- [AArch32_SystemAccessTrapSyndrome](#)
- [AArch32_TakeHypTrapException](#)
- [AArch32_TakeUndefInstrException](#)
- [AArch32_Undefined](#)

G1.1.2.1 AArch32_Abort

```
// AArch32_Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

func AArch32_Abort(fault : FaultRecord)
begin
    if fault.accessdesc.acctype == AccessType_IFETCH then
        AArch32_TakePrefetchAbortException(fault);
    else
        AArch32_TakeDataAbortException(fault);
    end;
end;
```

G1.1.2.2 AArch32_AbortSyndrome

```
// AArch32_AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort exceptions
// taken to Hyp mode
// from an AArch32 translation regime.
func AArch32_AbortSyndrome(exceptype : Exception, fault : FaultRecord,
```



```

        target_el : bits(2)) => ExceptionRecord

begin
    var except : ExceptionRecord = ExceptionSyndrome(exceptype);

    except.syndrome.iss = AArch32_FaultSyndrome(exceptype, fault);
    if exceptype == Exception_Watchpoint then
        except.vaddress = fault.watchptinfo.vaddress;
    else
        except.vaddress = ZeroExtend{64}(fault.vaddress);
    end;

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.ipaddress = ZeroExtend{NUM_PABITS}(fault.ipaddress);
    else
        except.ipavalid = FALSE;
    end;

    return except;
end;

```

G1.1.2.3 AArch32_CheckPCAlignment

```

// AArch32_CheckPCAlignment()
// =====

func AArch32_CheckPCAlignment()
begin
    let pc : bits(32) = ThisInstrAddr{}();

    if (CurrentInstrSet() == InstrSet_A32 && pc[1] == '1' || pc[0] == '1' then
        let accdesc : AccessDescriptor = CreateAccDescIFetch();
        var fault : FaultRecord = NoFault(accdesc, ZeroExtend{64}(pc));
        // Generate an Alignment fault Prefetch Abort exception
        fault.statuscode = Fault_Alignment;
        AArch32_Abort(fault);
    end;
end;

```

G1.1.2.4 AArch32_CommonFaultStatus

```

// AArch32_CommonFaultStatus()
// =====
// Return the common part of the fault status on reporting a Data
// or Prefetch Abort.

func AArch32_CommonFaultStatus(fault : FaultRecord, long_format : boolean) => bits(32)
begin
    var target : bits(32) = Zeros{};
    if IsExternalAbort(fault) then target[12] = fault.extflag; end; // ExT
    assert long_format == TRUE;
    target[9] = '1'; // LPAE
    target[5:0] = EncodeLDFSC(fault.statuscode, fault.level); // STATUS
    return target;
end;

```

G1.1.2.5 AArch32_ReportDataAbort

```

// AArch32_ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

func AArch32_ReportDataAbort(route_to_monitor : boolean, fault : FaultRecord)
begin

```

```

let long_format : boolean = TRUE;
assert route_to_monitor == FALSE;
var syndrome : bits(32) = AArch32_CommonFaultStatus(fault, long_format);

// bits of syndrome that are not common to I and D side
if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
    syndrome[13] = '1'; // CM
    syndrome[11] = '1'; // WnR
else
    syndrome[11] = if fault.write then '1' else '0'; // WnR
end;

if !long_format then
    syndrome[7:4] = fault.domain; // Domain
end;

if fault.accessdesc.acctype == AccessType_IC then
    var i_syndrome : bits(32);
    i_syndrome = ARBITRARY : bits(32);
    IFSR() = i_syndrome;
end;

DFSR() = syndrome;
DFAR() = fault.vaddress[31:0];
end;

```

G1.1.2.6 AArch32_ReportPrefetchAbort

```

// AArch32_ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

func AArch32_ReportPrefetchAbort(route_to_monitor : boolean, fault : FaultRecord)
begin
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    let long_format : boolean = TRUE;

    let fsr : bits(32) = AArch32_CommonFaultStatus(fault, long_format);
    IFSR() = fsr;
    IFAR() = fault.vaddress[31:0];
    return;
end;

```

G1.1.2.7 AArch32_TakeDataAbortException

```

// AArch32_TakeDataAbortException()
// =====

func AArch32_TakeDataAbortException(fault : FaultRecord)
begin
    let route_to_monitor : boolean = EffectiveEA() == '1' && IsExternalAbort(fault);
    let route_to_hyp : boolean = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR().TGE == '1' ||
        (IsDebugException(fault) && HDCR().TDE == '1') ||
        IsSecondStage(fault)));

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;

```

```

let lr_offset : integer = 8;

if IsDebugException(fault) then DBGDSCRext().MOE = fault.debugmoe; end;
if route_to_monitor then
    AArch32_ReportDataAbort(route_to_monitor, fault);
elseif PSTATE.EL == EL2 || route_to_hyp then
    let except : ExceptionRecord = AArch32_AbortSyndrome(Exception_DataAbort, fault, EL2);
    if PSTATE.EL == EL2 then
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
    end;
else
    AArch32_ReportDataAbort(route_to_monitor, fault);
    AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
end;
end;

```

G1.1.2.8 AArch32_TakePrefetchAbortException

```

// AArch32_TakePrefetchAbortException()
// =====

func AArch32_TakePrefetchAbortException(fault : FaultRecord)
begin

    let route_to_monitor : boolean = EffectiveEA() == '1' && IsExternalAbort(fault);
    let route_to_hyp : boolean = (EL2Enabled() && PSTATE.EL IN {EL0, EL1}) &&
        (HCR().TGE == '1' ||
        (IsDebugException(fault) && HCR().TDE == '1') ||
        IsSecondStage(fault));

    var except : ExceptionRecord;
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x0C;
    let lr_offset : integer = 4;

    if IsDebugException(fault) then DBGDSCRext().MOE = fault.debugmoe; end;
    if route_to_monitor then
        AArch32_ReportPrefetchAbort(route_to_monitor, fault);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault_Alignment then // PC Alignment fault
            except = ExceptionSyndrome(Exception_PCAlignment);
            except.vaddress = ThisInstrAddr{64}();
        else
            except = AArch32_AbortSyndrome(Exception_InstructionAbort, fault, EL2);
        end;
        if PSTATE.EL == EL2 then
            AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    else
        AArch32_ReportPrefetchAbort(route_to_monitor, fault);
        AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
    end;
end;

```

G1.1.2.9 AArch32_TakePhysicalFIQException

```

// AArch32_TakePhysicalFIQException()
// =====

func AArch32_TakePhysicalFIQException()
begin

```

```

let route_to_monitor : boolean = FALSE;
let route_to_hyp : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
    (HCR().TGE == '1' || HCR().FMO == '1'));
let preferred_exception_return : bits(32) = ThisInstrAddr{}();
let vect_offset : integer = 0x1C;
let lr_offset : integer = 4;
if PSTATE.EL == EL2 || route_to_hyp then
    let except : ExceptionRecord = ExceptionSyndrome(Exception_FIQ);
    AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
else
    AArch32_EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
end;
end;
end;

```

G1.1.2.10 AArch32_TakePhysicalIRQException

```

// AArch32_TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

func AArch32_TakePhysicalIRQException()
begin

    let route_to_monitor : boolean = FALSE;
    let route_to_hyp : boolean = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR().TGE == '1' || HCR().IMO == '1'));
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x18;
    let lr_offset : integer = 4;
    if PSTATE.EL == EL2 || route_to_hyp then
        let except : ExceptionRecord = ExceptionSyndrome(Exception_IRQ);
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32_EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
end;

```

G1.1.2.11 AArch32_TakePhysicalSErrorException

```

// AArch32_TakePhysicalSErrorException()
// =====

func AArch32_TakePhysicalSErrorException(implicit_esb : boolean)
begin
    var masked : boolean;
    var target_el : bits(2);

    (masked, target_el) = PhysicalSErrorTarget();
    assert !masked;

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;
    let fault : FaultRecord = GetPendingPhysicalSError();
    let lr_offset : integer = 8;

    let except : ExceptionRecord = AArch32_AbortSyndrome(Exception_DataAbort, fault, target_el);
    let route_to_monitor : boolean = (target_el == EL3);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    end;

    case target_el of
        when EL2 =>

```

```

        if PSTATE.EL == EL2 then
            AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    when EL1 =>
        AArch32_ReportDataAbort(route_to_monitor, fault);
        AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
    otherwise =>
        unreachable;
end;
end;
end;

```

G1.1.2.12 AArch32_TakeVirtualFIQException

```

// AArch32_TakeVirtualFIQException()
// =====

func AArch32_TakeVirtualFIQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR().TGE == '0' && HCR().FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1

    let preferred_exception_return : bits(32)= ThisInstrAddr{}();
    let vect_offset : integer = 0x1C;
    let lr_offset : integer = 4;

    AArch32_EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
end;

```

G1.1.2.13 AArch32_TakeVirtualIRQException

```

// AArch32_TakeVirtualIRQException()
// =====

func AArch32_TakeVirtualIRQException()
begin
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR().TGE == '0' && HCR().IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x18;
    let lr_offset : integer = 4;

    AArch32_EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
end;

```

G1.1.2.14 AArch32_TakeVirtualSErrorException

```

// AArch32_TakeVirtualSErrorException()
// =====

func AArch32_TakeVirtualSErrorException()
begin

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR().TGE == '0' && HCR().AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x10;
    let lr_offset : integer = 8;

    let vaddress : bits(32) = ARBITRARY : bits(32);
    let parity : boolean = FALSE;

```

```

let fault : Fault      = Fault_AsyncExternal;
let level : integer    = ARBITRARY : integer;
var fsr : bits(32)     = Zeros{};
fsr[12] = ImpDefBit("Virtual External abort type");
fsr[9] = '1';
fsr[5:0] = EncodeLDFSC(fault, level);
DFSR() = fsr;
DFAR() = ARBITRARY : bits(32);
ClearPendingVirtualSError();
AArch32_EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
end;

```

G1.1.2.15 AArch32_SoftwareBreakpoint

```

// AArch32_SoftwareBreakpoint()
// =====

func AArch32_SoftwareBreakpoint(immediate : bits(16))
begin

    let accdesc : AccessDescriptor = CreateAccDescIFetch();
    var fault : FaultRecord = NoFault(accdesc, ARBITRARY : bits(64));

    fault.statuscode = Fault_Debug;
    fault.debugmoe   = DebugException_BKPT;

    AArch32_Abort(fault);
end;

```

G1.1.2.16 DebugException

```

// DebugException
// =====
// Reason codes for debug exceptions, taken to AArch32

constant DebugException_Breakpoint : bits(4) = '0001';
constant DebugException_BKPT : bits(4)      = '0011';
constant DebugException_VectorCatch : bits(4) = '0101';
constant DebugException_Watchpoint : bits(4) = '1010';

```

G1.1.2.17 AArch32_CheckAdvSIMDOrFPRegisterTraps

```

// AArch32_CheckAdvSIMDOrFPRegisterTraps()
// =====
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

func AArch32_CheckAdvSIMDOrFPRegisterTraps(reg : bits(4))
begin
    if PSTATE.EL == EL1 && EL2Enabled() then
        let tid0 : bit = HCR().TID0;
        let tid3 : bit = HCR().TID3;
        if ((tid0 == '1' && reg == '0000') ||
            (tid3 == '1' && reg IN {'0101', '0110', '0111'})) then
            AArch32_SystemAccessTrap(M32_Hyp, 0x8);
        end;
    end;
end;

```

G1.1.2.18 AArch32_ExceptionClass

```
// AArch32_ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in HSR

func AArch32_ExceptionClass(exceptype : Exception) => (integer,bit)
begin
    var il_is_valid : boolean = TRUE;
    var ec : integer;
    case exceptype of
        when Exception_Uncategorized =>          ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap =>                ec = 0x01;
        when Exception_CP15RRTTrap =>            ec = 0x03;
        when Exception_CP15RRTTrap =>            ec = 0x04;
        when Exception_CP14RTTrap =>             ec = 0x05;
        when Exception_CP14DTTrap =>             ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap =>    ec = 0x07;
        when Exception_FPIDTrap =>               ec = 0x08;
        when Exception_CP14RRTTrap =>            ec = 0x0C;
        when Exception_IllegalState =>           ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall =>         ec = 0x11;
        when Exception_HypervisorCall =>        ec = 0x12;
        when Exception_MonitorCall =>           ec = 0x13;
        when Exception_InstructionAbort =>
            ec = if PSTATE.EL == EL2 then 0x21 else 0x20;
            il_is_valid = FALSE;
        when Exception_PCAalignment =>          ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort =>             ec = if PSTATE.EL == EL2 then 0x25 else 0x24;
        when Exception_FPTrappedException =>    ec = 0x28;
        otherwise =>                            unreachable;
    end;

    var il : bit;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    end;

    return (ec,il);
end;
```

G1.1.2.19 AArch32_ReportHypEntry

```
// AArch32_ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

func AArch32_ReportHypEntry(except : ExceptionRecord)
begin

    let exceptype : Exception = except.exceptype;

    var (ec,il) : (integer, bit) = AArch32_ExceptionClass(exceptype);
    let iss : bits(25) = except.syndrome.iss;
    let iss2 : bits(24) = except.syndrome.iss2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss[24] == '0' then
        il = '1';
    end;

    HSR() = ec[5:0]::il::iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAalignment} then
```

```

        HIFAR() = except.vaddress[31:0];
        HDFAR() = ARBITRARY : bits(32);
    elsif exceptype == Exception_DataAbort then
        HIFAR() = ARBITRARY : bits(32);
        HDFAR() = except.vaddress[31:0];
    end;

    if except.ipavalid then
        HPFAR()[31:4] = except.ipaddress[39:12];
    else
        HPFAR()[31:4] = ARBITRARY : bits(28);
    end;

    return;
end;

```

G1.1.2.20 AArch32_ResetControlRegisters

```

// AArch32_ResetControlRegisters()
// =====
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

impdef func AArch32_ResetControlRegisters(cold_reset : boolean)
begin
    return;
end;

```

G1.1.2.21 AArch32_TakeReset

```

// AArch32_TakeReset()
// =====
// Reset into AArch32 state

func AArch32_TakeReset(cold_reset : boolean)
begin
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL2) then
        AArch32_WriteMode(M32_Hyp);
    else
        AArch32_WriteMode(M32_Svc);
    end;

    // Reset System registers in the coproc=0b111x encoding space
    // and other system components
    AArch32_ResetControlRegisters(cold_reset);
    FPEXC().EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.[A,I,F] = '111'; // All asynchronous exceptions masked
    PSTATE.IT = '00000000'; // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR().TE; // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR().EE; // Endianness: EE=0: little-endian, EE=1: big-endian.
    else
        PSTATE.T = SCTLR().TE; // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR().EE; // Endianness: EE=0: little-endian, EE=1: big-endian.
    end;
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers

```



```
// R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
// is impossible to return from a reset in an architecturally defined way.
AArch32_ResetGeneralRegisters();
AArch32_ResetSpecialRegisters();
ResetExternalDebugRegisters(cold_reset);

var rv : bits(32); // IMPLEMENTATION DEFINED reset vector
rv = RVBAR() [31:1]::'0';

// The reset vector must be correctly aligned
assert rv[0] == '0' && (PSTATE.T == '1' || rv[1] == '0');

let branch_conditional : boolean = FALSE;
EDPRSR().R = '0'; // Leaving Reset State.
BranchTo{32}(rv, BranchType_RESET, branch_conditional);
end;
```

G1.1.2.22 ExcVectorBase

```
// ExcVectorBase()
// =====

func ExcVectorBase() => bits(32)
begin
    return VBAR() [31:5]::Zeros{5};
end;
```

G1.1.2.23 AArch32_FPtrappedException

```
// AArch32_FPtrappedException()
// =====

func AArch32_FPtrappedException(accumulated_exceptions : bits(8))
begin

    AArch32_TakeUndefInstrException();
end;
```

G1.1.2.24 AArch32_CallHypervisor

```
// AArch32_CallHypervisor()
// =====
// Performs a HVC call

func AArch32_CallHypervisor(immediate : bits(16))
begin
    assert HaveEL(EL2);

    AArch32_TakeHVCEXception(immediate);
end;
```

G1.1.2.25 AArch32_CallSupervisor

```
// AArch32_CallSupervisor()
// =====
// Calls the Supervisor

func AArch32_CallSupervisor(immediate_in : bits(16))
begin
    var immediate : bits(16) = immediate_in;
    if CurrentCond() != '1110' then
        immediate = ARBITRARY : bits(16);
    end;
```

```
    AArch32_TakeSVCException(immediate);
end;
```

G1.1.2.26 AArch32_TakeHVCException

```
// AArch32_TakeHVCException()
// =====

func AArch32_TakeHVCException(immediate : bits(16))
begin
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32_ITAdvance();
    let preferred_exception_return : bits(32) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x08;

    var except : ExceptionRecord = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome.iss[15:0] = immediate;

    if PSTATE.EL == EL2 then
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
    end;
end;
```

G1.1.2.27 AArch32_TakeSVCException

```
// AArch32_TakeSVCException()
// =====

func AArch32_TakeSVCException(immediate : bits(16))
begin

    AArch32_ITAdvance();
    let route_to_hyp : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';

    let preferred_exception_return : bits(32) = NextInstrAddr{}();
    let vect_offset : integer{} = 0x08;
    let lr_offset : integer{} = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        var except : ExceptionRecord = ExceptionSyndrome(Exception_SupervisorCall);
        except.syndrome.iss[15:0] = immediate;
        if PSTATE.EL == EL2 then
            AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
        end;
    else
        AArch32_EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
    end;
end;
```

G1.1.2.28 AArch32_EnterHypMode

```
// AArch32_EnterHypMode()
// =====
// Take an exception to Hyp mode.

noreturn func AArch32_EnterHypMode(except : ExceptionRecord, preferred_exception_return : bits(32),
                                   vect_offset : integer)
begin
    SynchronizeContext();
```

```

assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

if Halted() then
    AArch32_EnterHypModeInDebugState(except);
end;
let spsr : bits(32) = GetPSRFromPSTATE{}(AArch32_NonDebugState);
if ! except.excepttype IN {Exception_IRQ, Exception_FIQ} then
    AArch32_ReportHypEntry(except);
end;
AArch32_WriteMode(M32_Hyp);
SPSR_curr() = spsr;
ELR_hyp() = preferred_exception_return;
PSTATE.T = HSCTLR().TE; // PSTATE.J is RES0
PSTATE.SS = '0';
PSTATE.A = '1';
PSTATE.I = '1';
PSTATE.F = '1';
PSTATE.E = HSCTLR().EE;
PSTATE.IL = '0';
PSTATE.IT = '00000000';
let branch_conditional : boolean = FALSE;
BranchTo{32}(HVBAR() [31:5]::vect_offset[4:0], BranchType_EXCEPTION, branch_conditional);

CheckExceptionCatch(); // Check for debug event on exception entry

EndOfInstruction(EndOfInstructionReason_Exception);
end;

```

G1.1.2.29 AArch32_EnterMode

```

// AArch32_EnterMode()
// =====
// Take an exception to a mode other than Monitor and Hyp mode.

noreturn func AArch32_EnterMode(target_mode : bits(5),
                                preferred_exception_return : bits(32),
                                lr_offset : integer,
                                vect_offset : integer)
begin
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if Halted() then
        AArch32_EnterModeInDebugState(target_mode);
    end;
    let spsr : bits(32) = GetPSRFromPSTATE{}(AArch32_NonDebugState);
    AArch32_WriteMode(target_mode);
    SPSR_curr() = spsr;
    R(14) = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR().TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.[A,I,F] = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.[A,I] = '11';
    else
        PSTATE.I = '1';
    end;
    PSTATE.E = SCTLR().EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    let branch_conditional : boolean = FALSE;
    BranchTo{32}(ExcVectorBase() [31:5]::vect_offset[4:0], BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(); // Check for debug event on exception entry

```

```
EndOfInstruction(EndOfInstructionReason_Exception);
end;
```

G1.1.2.30 AArch32_CheckAdvSIMDOrFPEEnabled

```
// AArch32_CheckAdvSIMDOrFPEEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

func AArch32_CheckAdvSIMDOrFPEEnabled(fpexc_check_in : boolean, advsimd : boolean)
begin
    var fpexc_check : boolean = fpexc_check_in;
    var cpacr_asedis = CPACR().ASEDIS;
    var cpacr_cp10 = CPACR().cp10;

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
        // Check if access disabled in NSACR
        if NSACR().NSASEDIS == '1' then cpacr_asedis = '1'; end;
        if NSACR().cp10 == '0' then cpacr_cp10 = '00'; end;
    end;

    if PSTATE.EL != EL2 then
        // Check if Advanced SIMD disabled in CPACR
        if advsimd && cpacr_asedis == '1' then AArch32_Undefined(); end;

        // Check if access disabled in CPACR
        var disabled : boolean;
        case cpacr_cp10 of
            when '00' => disabled = TRUE;
            when '01' => disabled = PSTATE.EL == EL0;
            when '10' => disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
            when '11' => disabled = FALSE;
        end;
        if disabled then AArch32_Undefined(); end;
    end;

    if fpexc_check && FPEXC().EN == '0' then
        AArch32_Undefined();
    end;

    AArch32_CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
end;
```

G1.1.2.31 AArch32_CheckFPAdvSIMDTrap

```
// AArch32_CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

func AArch32_CheckFPAdvSIMDTrap(advsimd : boolean)
begin
    let ss : SecurityState = CurrentSecurityState();
    if HaveEL(EL2) && ss != SS_Secure then
        var hcptr_tase = HCPTR().TASE;
        var hcptr_cp10 = HCPTR().TCP10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            // Check if access disabled in NSACR
            if NSACR().NSASEDIS == '1' then hcptr_tase = '1'; end;
            if NSACR().cp10 == '0' then hcptr_cp10 = '1'; end;
        end;

        // Check if access disabled in HCPTR
        if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
            var except : ExceptionRecord = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            except.syndrome.iss[24:20] = ConditionSyndrome();
        end;
    end;
end;
```

```

        if advsimd then
            except.syndrome.iss[5] = '1';
        else
            except.syndrome.iss[5] = '0';
            except.syndrome.iss[3:0] = '1010';           // coproc field, always 0xA
        end;

        if PSTATE.EL == EL2 then
            AArch32_TakeUndefInstrException(except);
        else
            AArch32_TakeHypTrapException(except);
        end;
    end;
end;
end;

```

G1.1.2.32 AArch32_CheckForSMCUndefOrTrap

```

// AArch32_CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

func AArch32_CheckForSMCUndefOrTrap()
begin
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        Undefined();
    end;
end;

```

G1.1.2.33 AArch32_CheckForWFXTrap

```

// AArch32_CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

func AArch32_CheckForWFXTrap(target_el : bits(2), wfxtype : WFXType)
begin
    assert HaveEL(target_el);

    let is_wfe : boolean = wfxtype == WFXType_WFE;
    var trap : boolean;
    case target_el of
        when EL1 =>
            trap = (if is_wfe then SCTLRL().nTWE else SCTLRL().nTWI) == '0';
        when EL2 =>
            trap = (if is_wfe then HCR().TWE else HCR().TWI) == '1';
    end;

    if trap then
        if target_el == EL2 then
            var except : ExceptionRecord = ExceptionSyndrome(Exception_WFXTrap);
            except.syndrome.iss[24:20] = ConditionSyndrome();
            except.syndrome.iss[0] = if wfxtype == WFXType_WFI then '0' else '1';
            AArch32_TakeHypTrapException(except);
        else
            AArch32_TakeUndefInstrException();
        end;
    end;
end;

```

G1.1.2.34 AArch32_CheckITEnabled

```
// AArch32_CheckITEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

func AArch32_CheckITEnabled(mask : bits(4))
begin
    var it_disabled : bit;

    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR().ITD;
    else
        it_disabled = SCTLR().ITD;
    end;

    if it_disabled == '1' then
        if mask != '1000' then Undefined(); end;

        let accdesc : AccessDescriptor = CreateAccDescIFetch();
        let aligned = TRUE;
        // Otherwise whether the IT block is allowed depends on hwl of the next instruction.
        let next_instr = AArch32_MemSingle{16}(NextInstrAddr{32}(), accdesc, aligned);

        if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                        '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            Undefined();
        end;
    end;

return;
end;
```

G1.1.2.35 AArch32_CheckIllegalState

```
// AArch32_CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

func AArch32_CheckIllegalState()
begin
    if PSTATE.IL == '1' then
        let route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';

        let preferred_exception_return : bits(32) = ThisInstrAddr{}();
        let vect_offset : integer = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            let except : ExceptionRecord = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
            end;
        else
            AArch32_TakeUndefInstrException();
        end;
    end;

end;
```

G1.1.2.36 AArch32_CheckSETENDEnabled

```
// AArch32_CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

func AArch32_CheckSETENDEnabled()
begin
    var setend_disabled : bit;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR().SED;
    else
        setend_disabled = SCTLR().SED;
    end;
    if setend_disabled == '1' then
        Undefined();
    end;

    return;
end;
```

G1.1.2.37 AArch32_SystemAccessTrap

```
// AArch32_SystemAccessTrap()
// =====
// Trapped AArch32 System register access.

func AArch32_SystemAccessTrap(mode : bits(5), ec : integer)
begin
    let (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        let except : ExceptionRecord = AArch32_SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32_TakeHypTrapException(except);
    else
        AArch32_TakeUndefInstrException();
    end;
end;
```

G1.1.2.38 AArch32_SystemAccessTrapSyndrome

```
// AArch32_SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

func AArch32_SystemAccessTrapSyndrome(instr : bits(32), ec : integer) => ExceptionRecord
begin
    var except : ExceptionRecord;

    case ec of
        when 0x0 => except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3 => except = ExceptionSyndrome(Exception_CP15RTTTrap);
        when 0x4 => except = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5 => except = ExceptionSyndrome(Exception_CP14RTTTrap);
        when 0x6 => except = ExceptionSyndrome(Exception_CP14DTTTrap);
        when 0x7 => except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8 => except = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC => except = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise => unreachable;
    end;

    var iss : bits(20) = Zeros{};

    if except.exceptype == Exception_Uncategorized then
```

```

        return except;
    elsif except.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap,
                               Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss[13:10] = instr[19:16];      // CRn, Reg in case of VMRS
        iss[8:5]   = instr[15:12];      // Rt
        iss[9]     = '0';               // RES0

        if except.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
            iss[19:17] = instr[7:5];      // opc2
            iss[16:14] = instr[23:21];    // opc1
            iss[4:1]   = instr[3:0];      // CRm
        else //VMRS Access
            iss[19:17] = '000';          // opc2 - Hardcoded for VMRS
            iss[16:14] = '111';          // opc1 - Hardcoded for VMRS
            iss[4:1]   = '0000';         // CRm - Hardcoded for VMRS
        end;
    elsif except.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
                               Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss[19:16] = instr[7:4];          // opc1
        iss[13:10] = instr[19:16];        // Rt2
        iss[8:5]   = instr[15:12];        // Rt
        iss[4:1]   = instr[3:0];          // CRm
    elsif except.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss[19:12] = instr[7:0];          // imm8
        iss[4]     = instr[23];           // U
        iss[2:1]   = instr[24,21];        // P, W
        if instr[19:16] == '1111' then // Rn==15, LDC(Literal addressing)/STC
            iss[8:5] = ARBITRARY : bits(4);
            iss[3]   = '1';
        end;
    end;
    iss[0] = instr[20];                  // Direction

    except.syndrome.iss[24:20] = ConditionSyndrome();
    except.syndrome.iss[19:0] = iss;

    return except;
end;

```

G1.1.2.39 AArch32_TakeHypTrapException

```

// AArch32_TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

func AArch32_TakeHypTrapException(ec : integer)
begin
    let except : ExceptionRecord = AArch32_SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32_TakeHypTrapException(except);
end;

// AArch32_TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

func AArch32_TakeHypTrapException(except : ExceptionRecord)
begin
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x14;

    AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);

```


end;

G1.1.2.40 AArch32_TakeUndefInstrException

```
// AArch32_TakeUndefInstrException()
// =====

noreturn func AArch32_TakeUndefInstrException(except : ExceptionRecord)
begin

    let route_to_hyp : boolean = PSTATE.EL == EL0 && EL2Enabled() && HCR().TGE == '1';
    let preferred_exception_return : bits(32) = ThisInstrAddr{}();
    let vect_offset : integer = 0x04;
    let lr_offset : integer = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32_EnterHypMode(except, preferred_exception_return, vect_offset);
    elseif route_to_hyp then
        AArch32_EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32_EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
    end;
end;

// AArch32_TakeUndefInstrException()
// =====

noreturn func AArch32_TakeUndefInstrException()
begin
    let except : ExceptionRecord = ExceptionSyndrome(Exception_Uncategorized);
    AArch32_TakeUndefInstrException(except);
end;
```

G1.1.2.41 AArch32_Undefined

```
// AArch32_Undefined()
// =====

noreturn func AArch32_Undefined()
begin

    AArch32_TakeUndefInstrException();
end;
```

G1.1.3 aarch32/functions

This section includes the following pseudocode functions:

- [AArch32_DomainValid](#)
- [AArch32_FaultSyndrome](#)
- [AArch32_InstructionSyndromeValid](#)
- [A32ExpandImm](#)
- [A32ExpandImm_C](#)
- [DecodeImmShift](#)
- [DecodeRegShift](#)
- [RRX](#)
- [RRX_C](#)
- [SRTYPE](#)
- [Shift](#)

- [Shift_C](#)
- [T32ExpandImm](#)
- [T32ExpandImm_C](#)
- [VBitOps](#)
- [VCGType](#)
- [VCGType](#)
- [VFPNegMul](#)
- [AArch32_CheckCP15InstrCoarseTraps](#)
- [AArch32_ExclusiveMonitorsPass](#)
- [AArch32_IsExclusiveVA](#)
- [AArch32_MarkExclusiveVA](#)
- [AArch32_SetExclusiveMonitors](#)
- [CheckAdvSIMDEnabled](#)
- [CheckAdvSIMDOrVFPEEnabled](#)
- [CheckCryptoEnabled32](#)
- [CheckVFPEEnabled](#)
- [FPHalvedSub](#)
- [FPRSqrtStep](#)
- [FPrecipStep](#)
- [StandardFPCR](#)
- [AArch32_MemSingle](#)
- [AArch32_MemSingleRead](#)
- [AArch32_MemSingleWrite](#)
- [AArch32_UnalignedAccessFaults](#)
- [Hint_PreloadData](#)
- [Hint_PreloadDataForWrite](#)
- [Hint_PreloadInstr](#)
- [MemA](#)
- [MemO](#)
- [MemS](#)
- [MemU](#)
- [MemU_unpriv](#)
- [Mem_with_type](#)
- [AArch32_ResetGeneralRegisters](#)
- [AArch32_ResetSIMDFPRegisters](#)
- [AArch32_ResetSpecialRegisters](#)
- [AArch32_ResetSystemRegisters](#)
- [ALUExceptionReturn](#)
- [ALUWritePC](#)
- [BXWritePC](#)
- [BranchWritePC](#)
- [CBWritePC](#)
- [D](#)
- [Din](#)
- [H](#)
- [LR](#)
- [LoadWritePC](#)

- [LookUpRIndex](#)
- [Monitor](#)
- [PC32](#)
- [PCStoreValue](#)
- [PRBAR_EL](#)
- [PRLAR_EL](#)
- [Q](#)
- [Qin](#)
- [R](#)
- [RBankSelect](#)
- [ReadAnyAllocatedRegister](#)
- [ReadAnyAllocatedSPSR](#)
- [Rmode](#)
- [S](#)
- [WriteAnyAllocatedRegister](#)
- [WriteAnyAllocatedSPSR](#)
- [_Dclone](#)
- [AArch32_ExceptionReturn](#)
- [AArch32_ExecutingCPI0or11Instr](#)
- [AArch32_ITAdvance](#)
- [AArch32_InterruptPending](#)
- [AArch32_SysRegRead](#)
- [AArch32_SysRegRead64](#)
- [AArch32_SysRegReadCanWriteAPSR](#)
- [AArch32_SysRegWrite](#)
- [AArch32_SysRegWrite64](#)
- [AArch32_SysRegWriteM](#)
- [AArch32_WriteMode](#)
- [AArch32_WriteModeByInstr](#)
- [BadMode](#)
- [BankedRegisterAccessValid](#)
- [CPSRWriteByInstr](#)
- [ConditionPassed](#)
- [CurrentCond](#)
- [InITBlock](#)
- [LastInITBlock](#)
- [SPSRWriteByInstr](#)
- [SPSRaccessValid](#)
- [SelectInstrSet](#)
- [AArch32_DTLBI_ALL](#)
- [AArch32_DTLBI_ASID](#)
- [AArch32_DTLBI_VA](#)
- [AArch32_EffectiveBroadcast](#)
- [AArch32_ITLBI_ALL](#)
- [AArch32_ITLBI_ASID](#)
- [AArch32_ITLBI_VA](#)
- [AArch32_TLBI_ALL](#)

- [AArch32_TLBI_ASID](#)
- [AArch32_TLBI_IPAS2](#)
- [AArch32_TLBI_VA](#)
- [AArch32_TLBI_VAA](#)
- [AArch32_TLBI_VMALL](#)
- [AArch32_TLBI_VMALLS12](#)
- [Sat](#)

G1.1.3.1 AArch32_DomainValid

```
// AArch32_DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

func AArch32_DomainValid(statuscode : Fault, level : integer) => boolean
begin
    assert statuscode != Fault_None;

    case statuscode of
        when Fault_Domain =>
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk =>
            return level == 2;
        otherwise =>
            return FALSE;
    end;
end;
```

G1.1.3.2 AArch32_FaultSyndrome

```
// AArch32_FaultSyndrome()
// =====
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to AArch32 Hyp mode.

func AArch32_FaultSyndrome(exceptype : Exception, fault : FaultRecord) => bits(25)
begin
    assert fault.statuscode != Fault_None;

    var isstype : IssType;
    isstype.iss = Zeros{25};

    let d_side : boolean = exceptype == Exception_DataAbort;
    if d_side then

        if AArch32_InstructionSyndromeValid(fault) then
            isstype.iss[24:14] = LSInstructionSyndrome();
        end;

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss[8] = '1';
        end;

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss[6] = '1';
        elsif fault.statuscode == Fault_Exclusive then
            isstype.iss[6] = ARBITRARY : bit;
        else
            isstype.iss[6] = if fault.write then '1' else '0';
        end;

    end;

    if IsExternalAbort(fault) then isstype.iss[9] = fault.extflag; end;
```

```
isstype.iss[7] = if fault.s2fslwalk then '1' else '0';
isstype.iss[5:0] = EncodeLDFSC(fault.statuscode, fault.level);

return isstype.iss;
end;
```

G1.1.3.3 AArch32_InstructionSyndromeValid

```
// AArch32_InstructionSyndromeValid()
// =====
// Returns TRUE if ESR_ELx.ISV is '1' for the given Fault.

func AArch32_InstructionSyndromeValid(fault : FaultRecord) => boolean
begin
    if IsSecondStage(fault) && !fault.s2fslwalk then
        return (!IsExternalSyncAbort(fault) ||
                (IsExternalAbortOnWalk(fault) &&
                 ImpDefBool("ISV on second stage translation table walk")));
    end;

    return FALSE;
end;
```

G1.1.3.4 A32ExpandImm

```
// A32ExpandImm()
// =====

func A32ExpandImm(imm12 : bits(12)) => bits(32)
begin
    // PSTATE.C argument to following function call does not affect the imm32 result.
    let (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
end;
```

G1.1.3.5 A32ExpandImm_C

```
// A32ExpandImm_C()
// =====

func A32ExpandImm_C(imm12 : bits(12), carry_in : bit) => (bits(32), bit)
begin
    let unrotated_value : bits(32) = ZeroExtend()(imm12[7:0]);
    let (imm32, carry_out) : (bits(32), bit) = Shift_C{32}(unrotated_value, SRTYPE_ROR,
                                                            2*UInt(imm12[11:8]), carry_in);

    return (imm32, carry_out);
end;
```

G1.1.3.6 DecodeImmShift

```
// DecodeImmShift()
// =====

func DecodeImmShift(srtype : bits(2), imm5 : bits(5)) => (SRTYPE, integer{0..32})
begin
    var shift_t : SRTYPE;
    var shift_n : integer{0..32};
    case srtype of
        when '00' =>
            shift_t = SRTYPE_LSL; shift_n = UInt(imm5);
        when '01' =>
            shift_t = SRTYPE_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
    end;
```

```
        when '10' =>
            shift_t = SRTYPE_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11' =>
            if imm5 == '00000' then
                shift_t = SRTYPE_RRX;  shift_n = 1;
            else
                shift_t = SRTYPE_ROR;  shift_n = UInt(imm5);
            end;
        end;

    end;

    return (shift_t, shift_n);
end;
```

G1.1.3.7 DecodeRegShift

```
// DecodeRegShift()
// =====

func DecodeRegShift(srtype : bits(2)) => SRTYPE
begin
    var shift_t : SRTYPE;
    case srtype of
        when '00' => shift_t = SRTYPE_LSL;
        when '01' => shift_t = SRTYPE_LSR;
        when '10' => shift_t = SRTYPE_ASR;
        when '11' => shift_t = SRTYPE_ROR;
    end;
    return shift_t;
end;
```

G1.1.3.8 RRX

```
// RRX()
// =====

func RRX{N}(x : bits(N), carry_in : bit) => bits(N)
begin
    let (result, -) = RRX_C{N}(x, carry_in);
    return result;
end;
```

G1.1.3.9 RRX_C

```
// RRX_C()
// =====

func RRX_C{N}(x : bits(N), carry_in : bit) => (bits(N), bit)
begin
    let result : bits(N) = carry_in :: x[N-1:1];
    let carry_out : bit = x[0];
    return (result, carry_out);
end;
```

G1.1.3.10 SRTYPE

```
// SRTYPE
// =====

type SRTYPE of enumeration {SRTYPE_LSL, SRTYPE_LSR, SRTYPE_ASR, SRTYPE_ROR, SRTYPE_RRX};
```

G1.1.3.11 Shift

```
// Shift()
// =====

func Shift{N}(value : bits(N), srtype : SRTYPE, amount : integer, carry_in : bit) => bits(N)
begin
    let (result, -) = Shift_C{N}(value, srtype, amount, carry_in);
    return result;
end;
```

G1.1.3.12 Shift_C

```
// Shift_C()
// =====

func Shift_C{N}(value : bits(N), srtype : SRTYPE, amount : integer,
                carry_in : bit) => (bits(N), bit)
begin
    assert !(srtype == SRTYPE_RRX && amount != 1);

    var result : bits(N);
    var carry_out : bit;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRTYPE_LSL =>
                (result, carry_out) = LSL_C(value, amount);
            when SRTYPE_LSR =>
                (result, carry_out) = LSR_C(value, amount);
            when SRTYPE_ASR =>
                (result, carry_out) = ASR_C(value, amount);
            when SRTYPE_ROR =>
                (result, carry_out) = ROR_C(value, amount);
            when SRTYPE_RRX =>
                (result, carry_out) = RRX_C{N}(value, carry_in);
        end;
    end;

    return (result, carry_out);
end;
```

G1.1.3.13 T32ExpandImm

```
// T32ExpandImm()
// =====

func T32ExpandImm(imm12 : bits(12)) => bits(32)
begin
    // PSTATE.C argument to following function call does not affect the imm32 result.
    let (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
end;
```

G1.1.3.14 T32ExpandImm_C

```
// T32ExpandImm_C()
// =====

func T32ExpandImm_C(imm12 : bits(12), carry_in : bit) => (bits(32), bit)
begin
    var imm32 : bits(32);
    var carry_out : bit;
```

```

    if imm12[11:10] == '00' then
        case imm12[9:8] of
            when '00' =>
                imm32 = ZeroExtend{32}(imm12[7:0]);
            when '01' =>
                imm32 = '00000000' :: imm12[7:0] :: '00000000' :: imm12[7:0];
            when '10' =>
                imm32 = imm12[7:0] :: '00000000' :: imm12[7:0] :: '00000000';
            when '11' =>
                imm32 = imm12[7:0] :: imm12[7:0] :: imm12[7:0] :: imm12[7:0];
        end;
        carry_out = carry_in;
    else
        let unrotated_value : bits(32) = ZeroExtend{}('1'::imm12[6:0]);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12[11:7]));
    end;

    return (imm32, carry_out);
end;

```

G1.1.3.15 VBitOps

```

// VBitOps
// =====

type VBitOps of enumeration {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

```

G1.1.3.16 VCGEType

```

// VCGEType
// =====

type VCGEType of enumeration {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};

```

G1.1.3.17 VCGTtype

```

// VCGTtype
// =====

type VCGTtype of enumeration {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

```

G1.1.3.18 VFPNegMul

```

// VFPNegMul
// =====

type VFPNegMul of enumeration {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

```

G1.1.3.19 AArch32_CheckCP15InstrCoarseTraps

```

// AArch32_CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

func AArch32_CheckCP15InstrCoarseTraps(CRn : integer, nreg : integer, CRm : integer)
begin
    let trapped_encoding = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8  }) ||
                           (CRn == 10 && CRm IN {0,1,    4,    8  }) ||
                           (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then

```



```

    let major : integer = if nreg == 1 then CRn else CRm;
    // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR().CRn/HSTR().CRm
    // and MRC and MCR disabled by HCR.TIDCP.
    if ((! major IN {4,14} && HSTR()[major] == '1') ||
        (HCR().TIDCP == '1' && nreg == 1 && trapped_encoding)) then
        if (PSTATE.EL == EL0 &&
            ImpDefBool("UNDEF unallocated CP15 access at EL0")) then
            Undefined();
        end;
        AArch32_SystemAccessTrap(M32_Hyp, 0x3);
    end;
end;
end;
end;

```

G1.1.3.20 AArch32_ExclusiveMonitorsPass

```

// AArch32_ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give a memory abort.

func AArch32_ExclusiveMonitorsPass(address : bits(32),
    size : integer{1, 2, 4, 8, 16, 32}) => boolean
begin
    let acqrel : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged);
    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    if !AArch32_IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;
    end;

    let memaddrdesc : AddressDescriptor = AArch32_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32_Abort(memaddrdesc.fault);
    end;

    var passed : boolean = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    return passed;
end;

```

G1.1.3.21 AArch32_IsExclusiveVA

```
// AArch32_IsExclusiveVA()
// =====
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

impdef func AArch32_IsExclusiveVA(address : bits(32),
                                processorid : integer,
                                size : integer) => boolean

begin
    return TRUE;
end;
```

G1.1.3.22 AArch32_MarkExclusiveVA

```
// AArch32_MarkExclusiveVA()
// =====
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

func AArch32_MarkExclusiveVA(address : bits(32), processorid : integer, size : integer)

begin
    return;
end;
```

G1.1.3.23 AArch32_SetExclusiveMonitors

```
// AArch32_SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

func AArch32_SetExclusiveMonitors(address : bits(32), size : integer{1, 2, 4, 8, 16, 32})

begin
    let acqrel : boolean = FALSE;
    let privileged : boolean = PSTATE.EL != EL0;
    let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged);
    let aligned : boolean = IsAlignedSize(address, size);

    if !aligned then
        let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
        AArch32_Abort(fault);
    end;

    let memaddrdesc : AddressDescriptor = AArch32_TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;
    end;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    end;

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32_MarkExclusiveVA(address, ProcessorID(), size);
end;
```

G1.1.3.24 CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====

func CheckAdvSIMDEnabled()
begin
    let fpexc_check : boolean = TRUE;
    let advsimd : boolean = TRUE;

    AArch32_CheckAdvSIMDOrFPEEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31 do
        _Dclone[[i]] = D(i);
    end;

    return;
end;
```

G1.1.3.25 CheckAdvSIMDOrVFPEEnabled

```
// CheckAdvSIMDOrVFPEEnabled()
// =====

func CheckAdvSIMDOrVFPEEnabled(include_fpexc_check : boolean, advsimd : boolean)
begin
    AArch32_CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
end;
```

G1.1.3.26 CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// =====

func CheckCryptoEnabled32()
begin
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
end;
```

G1.1.3.27 CheckVFPEEnabled

```
// CheckVFPEEnabled()
// =====

func CheckVFPEEnabled(include_fpexc_check : boolean)
begin
    let advsimd : boolean = FALSE;
    AArch32_CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
end;
```

G1.1.3.28 FPHalvedSub

```
// FPHalvedSub()
// =====

func FPHalvedSub{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    let rounding : FPRounding = FPRoundingMode(fpcr);
    let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean = (type1 == FType_Infinity);
        let inf2 : boolean = (type2 == FType_Infinity);
        let zero1 : boolean = (type1 == FType_Zero);
        let zero2 : boolean = (type2 == FType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN{N}(fpcr);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity{N}('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity{N}('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero{N}(sign1);
        else
            let result_value : real = (value1 - value2) / 2.0;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero{N}(result_sign);
            else
                result = FPRound{N}(result_value, fpcr);
            end;
        end;
    end;
    return result;
end;
```

G1.1.3.29 FPRSqrtStep

```
// FPRSqrtStep()
// =====

func FPRSqrtStep{N}(op1 : bits(N), op2 : bits(N)) => bits(N)
begin
    assert N == 32;
    let fpcr : FPCR_Type = StandardFPCR();
    let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean = (type1 == FType_Infinity);
        let inf2 : boolean = (type2 == FType_Infinity);
        let zero1 : boolean = (type1 == FType_Zero);
        let zero2 : boolean = (type2 == FType_Zero);
        var product : bits(N);
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero{N}('0');
        else
            product = FPMul{N}(op1, op2, fpcr);
        end;
        let three : bits(N) = FPThree{}('0');
        result = FPHalvedSub{N}(three, product, fpcr);
    end;
    return result;
end;
```

end;

G1.1.3.30 FPREcipStep

```
// FPREcipStep()
// =====

func FPREcipStep{N}(op1 : bits(N), op2 : bits(N)) => bits(N)
begin
    assert N == 32;
    let fpcr : FPCR_Type = StandardFPCR();
    let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean = (type1 == FType_Infinity);
        let inf2 : boolean = (type2 == FType_Infinity);
        let zero1 : boolean = (type1 == FType_Zero);
        let zero2 : boolean = (type2 == FType_Zero);
        var product : bits(N);
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero{N}('0');
        else
            product = FPMul{N}(op1, op2, fpcr);
        end;
        let two : bits(N) = FPTwo{}('0');
        result = FPSub{N}(two, product, fpcr);
    end;
    return result;
end;
```

G1.1.3.31 StandardFPCR

```
// StandardFPCR()
// =====

func StandardFPCR() => FPCR_Type
begin
    let value : bits(32) = '00000' :: FPSCR().AHP :: '110000000000000000000000000000';
    return ZeroExtend{64}(value);
end;
```

G1.1.3.32 AArch32_MemSingle

```
// AArch32_MemSingle
// =====

accessor AArch32_MemSingle{size : integer{8, 16, 32, 64}}(address : bits(32),
    accdesc : AccessDescriptor,
    aligned : boolean) <=> value : bits(size)

begin
    // Perform an atomic, little-endian read of 'size' bits.
    getter
        let bytes : integer{} = size DIV 8;
        var value : bits(size);
        var memaddrdesc : AddressDescriptor;
        var memstatus : PhysMemRetStatus;

        (value, memaddrdesc, memstatus) = AArch32_MemSingleRead{size}(address, accdesc, aligned);

        // Check for a fault from translation or the output of translation.
        if IsFault(memaddrdesc) then
            AArch32_Abort(memaddrdesc.fault);
        end;
```

```

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, bytes, accdesc);
    end;

    return value;
end;

// Perform an atomic, little-endian write of 'size' bits.
setter
    let bytes : integer{} = size DIV 8;
    var memaddrdesc : AddressDescriptor;
    var memstatus : PhysMemRetStatus;

    (memaddrdesc, memstatus) = AArch32_MemSingleWrite{size}(address, accdesc, aligned, value);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch32_Abort(memaddrdesc.fault);
    end;

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);
    end;

    return;
end;
end;

```

G1.1.3.33 AArch32_MemSingleRead

```

// AArch32_MemSingleRead()
// =====
// Perform an atomic, little-endian read of 'size' bits.

func AArch32_MemSingleRead{size : integer{8, 16, 32, 64}}(address : bits(32),
    accdesc_in : AccessDescriptor,
    aligned : boolean
    ) => (bits(size), AddressDescriptor,
    PhysMemRetStatus)

begin
    let bytes : integer{} = size DIV 8;
    var value : bits(size) = ARBITRARY : bits(size);
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;
    var accdesc : AccessDescriptor = accdesc_in;
    assert IsAlignedSize(address, bytes);

    var memaddrdesc : AddressDescriptor;
    memaddrdesc = AArch32_TranslateAddress(address, accdesc, aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (value, memaddrdesc, memstatus);
    end;

    // Memory array access
    (memstatus, value) = PhysMemRead{size}(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);
    end;

    if accdesc.acctype == AccessType_IFETCH then
        memaddrdesc.fault = AArch32_CheckDebug(address, accdesc, bytes);
    end;
end;

```

```
    return (value, memaddrdesc, memstatus);  
end;
```

G1.1.3.34 AArch32_MemSingleWrite

```
// AArch32_MemSingleWrite()  
// =====  
// Perform an atomic, little-endian write of 'size' bits.  
  
func AArch32_MemSingleWrite(size : integer{8, 16, 32, 64})(address : bits(32),  
    accdesc_in : AccessDescriptor,  
    aligned : boolean,  
    value : bits(size)  
    ) => (AddressDescriptor, PhysMemRetStatus)  
  
begin  
    let bytes : integer{} = size DIV 8;  
    var accdesc : AccessDescriptor = accdesc_in;  
    assert IsAlignedSize(address, bytes);  
  
    var memaddrdesc : AddressDescriptor;  
    var memstatus : PhysMemRetStatus = ARBITRARY : PhysMemRetStatus;  
    memaddrdesc = AArch32_TranslateAddress(address, accdesc, aligned, bytes);  
  
    // Check for aborts or debug exceptions  
    if IsFault(memaddrdesc) then  
        return (memaddrdesc, memstatus);  
    end;  
  
    // Effect on exclusives  
    if memaddrdesc.memattrs.shareability != Shareability_NSH then  
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);  
    end;  
  
    memstatus = PhysMemWrite(size)(memaddrdesc, accdesc, value);  
    if IsFault(memstatus) then  
        return (memaddrdesc, memstatus);  
    end;  
  
    return (memaddrdesc, memstatus);  
end;
```

G1.1.3.35 AArch32_UnalignedAccessFaults

```
// AArch32_UnalignedAccessFaults()  
// =====  
// Determine whether the unaligned access generates an Alignment fault  
  
func AArch32_UnalignedAccessFaults(accdesc : AccessDescriptor) => boolean  
begin  
    return (AlignmentEnforced() ||  
        accdesc.a32lsmd ||  
        accdesc.exclusive ||  
        accdesc.acqsc ||  
        accdesc.relsc);  
end;
```

G1.1.3.36 Hint_PreloadData

```
// Hint_PreloadData()  
// =====  
// Perform a preload data hint.  
  
impdef func Hint_PreloadData(address : bits(32))
```

```
begin
    return;
end;
```

G1.1.3.37 Hint_PreloadDataForWrite

```
// Hint_PreloadDataForWrite()
// =====
// Perform a preload data hint with a probability that the use will be for a write.

impdef func Hint_PreloadDataForWrite(address : bits(32))
begin
    return;
end;
```

G1.1.3.38 Hint_PreloadInstr

```
// Hint_PreloadInstr()
// =====
// Perform a preload instruction hint.

impdef func Hint_PreloadInstr(address : bits(32))
begin
    return;
end;
```

G1.1.3.39 MemA

```
// MemA - accessor
// =====

accessor MemA{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value : bits(size)
begin
    getter
        let acqrel : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged);
        return Mem_with_type{size}(address, accdesc);
    end;

    setter
        let acqrel : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let accdesc : AccessDescriptor = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged);
        Mem_with_type{size}(address, accdesc) = value;
    end;
end;
```

G1.1.3.40 MemO

```
// MemO
// ====

accessor MemO{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value : bits(size)
begin
    getter
        let acquire : boolean = TRUE;
        let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_LOAD, acquire);
        return Mem_with_type{size}(address, accdesc);
    end;

    setter
        let acquire : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescAcqRel(MemOp_STORE, acquire);
```



```

        Mem_with_type{size}(address, accdesc) = value;
    end;
end;

```

G1.1.3.41 MemS

```

// MemS - accessor
// =====

accessor MemS{size : integer{8, 16, 32, 64}}(address : bits(32),
                                              within8byte : boolean) <=> value : bits(size)
begin
    // Memory accessor for streaming load multiple instructions
    getter
        let accdesc : AccessDescriptor = CreateAccDescA32LSMD(MemOp_LOAD, within8byte);
        return Mem_with_type{size}(address, accdesc);
    end;

    // Memory accessor for streaming store multiple instructions
    setter
        let accdesc : AccessDescriptor = CreateAccDescA32LSMD(MemOp_STORE, within8byte);
        Mem_with_type{size}(address, accdesc) = value;
    end;
end;

```

G1.1.3.42 MemU

```

// MemU - accessor
// =====

accessor MemU{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value: bits(size)
begin
    getter
        let nontemporal : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged);
        return Mem_with_type{size}(address, accdesc);
    end;

    setter
        let nontemporal : boolean = FALSE;
        let privileged : boolean = PSTATE.EL != EL0;
        let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged);
        Mem_with_type{size}(address, accdesc) = value;
    end;
end;

```

G1.1.3.43 MemU_unpriv

```

// MemU_unpriv
// =====

accessor MemU_unpriv{size : integer{8, 16, 32, 64}}(address : bits(32)) <=> value : bits(size)
begin
    getter
        let nontemporal : boolean = FALSE;
        let privileged : boolean = FALSE;
        let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged);
        return Mem_with_type{size}(address, accdesc);
    end;

    setter
        let nontemporal : boolean = FALSE;
        let privileged : boolean = FALSE;

```

```

        let accdesc : AccessDescriptor = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged);
        Mem_with_type{size}(address, accdesc) = value;
    end;
end;

```

G1.1.3.44 Mem_with_type

```

// Mem_with_type
// =====

accessor Mem_with_type(size : integer{8, 16, 32, 64})(address : bits(32),
                                                    accdesc_in : AccessDescriptor
                                                    ) <=> value_in : bits(size)

begin
    // Perform a read of 'size' bits. The access byte order is reversed for a big-endian access.
    // Instruction fetches would call AArch32_MemSingle directly.
    getter
        let bytes : integer{} = size DIV 8;
        var accdesc : AccessDescriptor = accdesc_in;
        var value : bits(size);

        // Check alignment on size of element accessed, not overall access size
        var aligned : boolean;
        if accdesc.ispair then
            let half : integer{} = (size DIV 2) as integer{32};
            aligned = IsAlignedSize(address, half DIV 8);
        else
            aligned = IsAlignedSize(address, bytes);
        end;

        if !aligned && AArch32_UnalignedAccessFaults(accdesc) then
            let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
            AArch32_Abort(fault);
        end;

        if aligned then
            value = AArch32_MemSingle{size}(address, accdesc, aligned);
        else
            assert bytes > 1;
            value[7:0] = AArch32_MemSingle{8}(address, accdesc, aligned);

            // For subsequent bytes, if they cross to a new translation page which assigns
            // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
            // will generate an Alignment Fault.
            let c : Constraint = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE; end;

            for i = 1 to bytes-1 do
                value[i*8:] = AArch32_MemSingle{8}(address+i, accdesc, aligned);
            end;
        end;

        if BigEndian(accdesc.acctype) then
            value = BigEndianReverse{size}(value);
        end;

        return value;
    end;

    // Perform a write of 'size' bits. The byte order is reversed for a big-endian access.
    setter
        let bytes : integer{} = size DIV 8;
        var value : bits(size) = value_in;
        var accdesc : AccessDescriptor = accdesc_in;

        // Check alignment on size of element accessed, not overall access size

```

```

var aligned : boolean;
if accdesc.ispair then
    let half : integer{} = (size DIV 2) as integer{32};
    aligned = IsAlignedSize(address, half DIV 8);
else
    aligned = IsAlignedSize(address, bytes);
end;

if !aligned && AArch32_UnalignedAccessFaults(accdesc) then
    let fault : FaultRecord = AlignmentFault(accdesc, ZeroExtend{64}(address));
    AArch32_Abort(fault);
end;

if BigEndian(accdesc.acctype) then
    value = BigEndianReverse{size}(value);
end;

if aligned then
    AArch32_MemSingle{size}(address, accdesc, aligned) = value;
else
    assert bytes > 1;
    AArch32_MemSingle{8}(address, accdesc, aligned) = value[7:0];

    // For subsequent bytes, if they cross to a new translation page which assigns
    // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
    // will generate an Alignment Fault.

    let c : Constraint = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE; end;

    for i = 1 to bytes-1 do
        AArch32_MemSingle{8}(address+i, accdesc, aligned) = value[i*8];
    end;
end;
return;
end;
end;

```

G1.1.3.45 AArch32_ResetGeneralRegisters

```

// AArch32_ResetGeneralRegisters()
// =====

func AArch32_ResetGeneralRegisters()
begin

    for i = 0 to 7 do
        R(i) = ARBITRARY : bits(32);
    end;

    for i = 8 to 12 do
        Rmode(i, M32_User) = ARBITRARY : bits(32);
        Rmode(i, M32_FIQ) = ARBITRARY : bits(32);
    end;

    if HaveEL(EL2) then Rmode(13, M32_Hyp) = ARBITRARY : bits(32); end; // No R14_hyp
    for i = 13 to 14 do
        Rmode(i, M32_User) = ARBITRARY : bits(32);
        Rmode(i, M32_FIQ) = ARBITRARY : bits(32);
        Rmode(i, M32_IRQ) = ARBITRARY : bits(32);
        Rmode(i, M32_Svc) = ARBITRARY : bits(32);
        Rmode(i, M32_Abort) = ARBITRARY : bits(32);
        Rmode(i, M32_Undef) = ARBITRARY : bits(32);
        if HaveEL(EL3) then Rmode(i, M32_Monitor) = ARBITRARY : bits(32); end;
    end;

    return;
end;

```

G1.1.3.46 AArch32_ResetSIMDFPRegisters

```
// AArch32_ResetSIMDFPRegisters()
// =====

func AArch32_ResetSIMDFPRegisters()
begin

    for i = 0 to 15 do
        Q(i) = ARBITRARY : bits(128);
    end;

    return;
end;
```

G1.1.3.47 AArch32_ResetSpecialRegisters

```
// AArch32_ResetSpecialRegisters()
// =====

func AArch32_ResetSpecialRegisters()
begin

    // AArch32 special registers
    SPSR_fiq() [31:0] = ARBITRARY : bits(32);
    SPSR_irq() [31:0] = ARBITRARY : bits(32);
    SPSR_svc() [31:0] = ARBITRARY : bits(32);
    SPSR_abt() [31:0] = ARBITRARY : bits(32);
    SPSR_und() [31:0] = ARBITRARY : bits(32);
    if HaveEL(EL2) then
        SPSR_hyp() = ARBITRARY : bits(32);
        ELR_hyp() = ARBITRARY : bits(32);
    end;

    // External debug special registers
    DLR() = ARBITRARY : bits(32);
    DSPSR() = ARBITRARY : bits(32);

    return;
end;
```

G1.1.3.48 AArch32_ResetSystemRegisters

```
// AArch32_ResetSystemRegisters()
// =====

impdef func AArch32_ResetSystemRegisters(cold_reset : boolean)
begin
    return;
end;
```

G1.1.3.49 ALUEXceptionReturn

```
// ALUEXceptionReturn()
// =====

func ALUEXceptionReturn(address : bits(32))
begin
    if PSTATE.EL == EL2 then
        Undefined();
    elseif PSTATE.M IN {M32_User, M32_System} then
        let c : Constraint = ConstrainUnpredictable(Unpredictable_ALUEXCEPTIONRETURN);
```

```

        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF =>
                Undefined();
            when Constraint_NOP =>
                ExecuteAsNOP();
        end;
    else
        AArch32_ExceptionReturn(address, SPSR_curr());
    end;
end;

```

G1.1.3.50 ALUWritePC

```

// ALUWritePC()
// =====

func ALUWritePC(address : bits(32))
begin
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address, BranchType_INDIR);
    else
        BranchWritePC(address, BranchType_INDIR);
    end;
end;

```

G1.1.3.51 BXWritePC

```

// BXWritePC()
// =====

func BXWritePC(address_in : bits(32), branch_type : BranchType)
begin
    var address : bits(32) = address_in;
    if address[0] == '1' then
        SelectInstrSet(InstrSet_T32);
        address[0] = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the PE takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address[1] == '1' && ConstrainUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
            address[1] = '0';
        end;
    end;
    let branch_conditional : boolean = CurrentCond() != '111x';
    BranchTo{32}(address, branch_type, branch_conditional);
end;

```

G1.1.3.52 BranchWritePC

```

// BranchWritePC()
// =====

func BranchWritePC(address_in : bits(32), branch_type : BranchType)
begin
    var address : bits(32) = address_in;
    if CurrentInstrSet() == InstrSet_A32 then
        address[1:0] = '00';
    else
        address[0] = '0';
    end;
end;

```

```
    let branch_conditional : boolean = CurrentCond() != '111x';  
    BranchTo{32}(address, branch_type, branch_conditional);  
end;
```

G1.1.3.53 CBWritePC

```
// CBWritePC()  
// =====  
// Takes a branch from a CBNZ/CBZ instruction.  
  
func CBWritePC(address_in : bits(32))  
begin  
    var address : bits(32) = address_in;  
    assert CurrentInstrSet() == InstrSet_T32;  
    address[0] = '0';  
    let branch_conditional : boolean = TRUE;  
    BranchTo{32}(address, BranchType_DIR, branch_conditional);  
end;
```

G1.1.3.54 D

```
// D - accessor  
// =====  
  
accessor D(n : integer) <=> value : bits(64)  
begin  
    getter  
        assert n >= 0 && n <= 31;  
        let vreg : bits(128) = V{(n DIVRM 2)};  
        return vreg[(n MOD 2)*:64];  
    end;  
    setter  
        assert n >= 0 && n <= 31;  
        var vreg : bits(128) = V{(n DIVRM 2)};  
        vreg[(n MOD 2)*:64] = value;  
        V{128}(n DIVRM 2) = vreg;  
    end;  
end;
```

G1.1.3.55 Din

```
// Din  
// ===  
// Return the initial value of D input  
  
func Din(n : integer) => bits(64)  
begin  
    assert n >= 0 && n <= 31;  
    return _Dclone[[n]];  
end;
```

G1.1.3.56 H

```
// H - accessor  
// =====  
  
accessor H(n : integer) <=> value : bits(16)  
begin  
    getter  
        assert n >= 0 && n <= 31;  
        return S(n)[15:0];  
    end;  
    setter  
        S(n) = ZeroExtend{32}(value);  
end;
```

```
end;
end;
```

G1.1.3.57 LR

```
// LR - accessor
// =====

accessor LR() <=> value : bits(32)
begin
    getter
        return R(14);
    end;
    setter
        R(14) = value;
    end;
end;
```

G1.1.3.58 LoadWritePC

```
// LoadWritePC()
// =====

func LoadWritePC(address : bits(32))
begin
    BXWritePC(address, BranchType_INDIR);
end;
```

G1.1.3.59 LookUpRIndex

```
// LookUpRIndex()
// =====

func LookUpRIndex(n : integer, mode : bits(5)) => integer
begin
    assert n >= 0 && n <= 14;

    var result : integer;
    case n of // Select index by mode:    usr fiq irq svc abt und hyp
        when 8 => result = RBankSelect(mode, 8, 24, 8, 8, 8, 8, 8);
        when 9 => result = RBankSelect(mode, 9, 25, 9, 9, 9, 9, 9);
        when 10 => result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11 => result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12 => result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13 => result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14 => result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise => result = n;
    end;

    return result;
end;
```

G1.1.3.60 Monitor

```
// Monitor mode registers
// =====
// The Monitor mode registers do not map to X registers, so must be defined separately

var SP_mon : bits(32);

var LR_mon : bits(32);
```

G1.1.3.61 PC32

```
// PC32
// ====
// Read 32-bit program counter

func PC32() => bits(32)
begin
    return R(15);                // This includes the offset from AArch32 state
end;
```

G1.1.3.62 PCStoreValue

```
// PCStoreValue()
// =====

func PCStoreValue() => bits(32)
begin
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC32();
end;
```

G1.1.3.63 PRBAR_EL

```
// PRBAR_EL - accessor
// =====

accessor PRBAR_EL(EL : bits(2), index : integer) <=> value : PRBARnType
begin
    getter
        var r : bits(32);
        case EL of
            when EL1 => r = PRBARn[[index]];
            when EL2 => r = HPRBARn[[index]];
            otherwise => unreachable;
        end;
        return r;
    end;
    setter
        case EL of
            when EL1 => PRBARn[[index]] = value;
            when EL2 => HPRBARn[[index]] = value;
            otherwise => unreachable;
        end;
    end;
end;
```

G1.1.3.64 PRLAR_EL

```
// PRLAR_EL - accessor
// =====

accessor PRLAR_EL(EL : bits(2), index : integer) <=> value : PRLARnType
begin
    getter
        var r : bits(32);
        case EL of
            when EL1 => r = PRLARn[[index]];
            when EL2 => r = HPRLARn[[index]];
            otherwise => unreachable;
        end;
        return r;
    end;
end;
```



```

end;
setter
case EL of
when EL1 => PRLARn[[index]] = value;
when EL2 => HPRLARn[[index]] = value;
otherwise => unreachable;
end;
end;
end;

```

G1.1.3.65 Q

```

// Q - accessor
// =====

accessor Q(n : integer) <=> value : bits(128)
begin
getter
assert n >= 0 && n <= 15;
return V{128}(n);
end;
setter
assert n >= 0 && n <= 15;
V{128}(n) = value;
end;
end;

```

G1.1.3.66 Qin

```

// Qin
// ==
// Return the initial value of Q input

func Qin(n : integer) => bits(128)
begin
assert n >= 0 && n <= 15;
return Din(2*n+1)::Din(2*n);
end;

```

G1.1.3.67 R

```

// R - accessor
// =====

accessor R(n : integer) <=> value : bits(32)
begin
getter
if n == 15 then
let offset : integer{} = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
return _PC[31:0] + offset;
else
return Rmode(n, PSTATE.M);
end;
end;
setter
Rmode(n, PSTATE.M) = value;
return;
end;
end;

// R - accessor
// =====

accessor R(lr : integer, hr : integer) <=> value : bits(64)

```

```
begin
  getter
    return R(hr) :: R(lr);
  end;

  setter
    R(lr) = value[0+:32];
    R(hr) = value[32+:32];
  end;
end;
```

G1.1.3.68 RBankSelect

```
// RBankSelect()
// =====

func RBankSelect(mode : bits(5), usr : integer, fiq : integer, irq : integer,
                 svc : integer, abt : integer, und : integer, hyp : integer) => integer
begin

  var result : integer;
  case mode of
    when M32_User   => result = usr; // User mode
    when M32_FIQ    => result = fiq; // FIQ mode
    when M32_IRQ    => result = irq; // IRQ mode
    when M32_Svc    => result = svc; // Supervisor mode
    when M32_Abort  => result = abt; // Abort mode
    when M32_Hyp    => result = hyp; // Hyp mode
    when M32_Undef  => result = und; // Undefined mode
    when M32_System => result = usr; // System mode uses User mode registers
    otherwise       => unreachable; // Monitor mode
  end;

  return result;
end;
```

G1.1.3.69 ReadAnyAllocatedRegister

```
// ReadAnyAllocatedRegister()
// =====

impdef func ReadAnyAllocatedRegister() => bits(32)
begin
  return Zeros{32};
end;
```

G1.1.3.70 ReadAnyAllocatedSPSR

```
// ReadAnyAllocatedSPSR()
// =====

impdef func ReadAnyAllocatedSPSR() => bits(32)
begin
  return Zeros{32};
end;
```

G1.1.3.71 Rmode

```
// Rmode - accessor
// =====

accessor Rmode(n : integer, mode : bits(5)) <=> value : bits(32)
begin
  getter
```

```

    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor; end;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[[n]][31:0];
        end;
    else
        return _R[[LookUpRIndex(n, mode)]] [31:0];
    end;
end;

setter
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor; end;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[[n]][31:0] = value;
        end;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable_ZERoupper) then
            _R[[LookUpRIndex(n, mode)]] = ZeroExtend{64}(value);
        else
            _R[[LookUpRIndex(n, mode)]] [31:0] = value;
        end;
    end;
end;
end;

```

G1.1.3.72 S

```

// S - accessor
// =====

accessor S(n : integer) <=> value : bits(32)
begin
    getter
        assert n >= 0 && n <= 31;
        let vreg : bits(128) = V{}(n DIVRM 4);
        return vreg[(n MOD 4)*:32];
    end;

    setter
        assert n >= 0 && n <= 31;
        var vreg : bits(128) = V{}(n DIVRM 4);
        vreg[(n MOD 4)*:32] = value;
        V{128}(n DIVRM 4) = vreg;
    end;
end;

```

G1.1.3.73 WriteAnyAllocatedRegister

```
// WriteAnyAllocatedRegister()
// =====

impdef func WriteAnyAllocatedRegister(value : bits(32))
begin
    return;
end;
```

G1.1.3.74 WriteAnyAllocatedSPSR

```
// WriteAnyAllocatedSPSR()
// =====

impdef func WriteAnyAllocatedSPSR(value : bits(32))
begin
    return;
end;
```

G1.1.3.75 _Dclone

```
// _Dclone[]
// =====
// Clone the 64-bit Advanced SIMD and VFP extension register bank for use as input to
// instruction pseudocode, to avoid read-after-write for Advanced SIMD and VFP operations.

var _Dclone : array [[32]] of bits(64);
```

G1.1.3.76 AArch32_ExceptionReturn

```
// AArch32_ExceptionReturn()
// =====

func AArch32_ExceptionReturn(new_pc_in : bits(32), spsr : bits(32))
begin
    var new_pc : bits(32) = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR{32}(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc[1:0] = ARBITRARY : bits(2);
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc[0] = '0'; // T32
        else
            new_pc[1:0] = '00'; // A32
        end;
    end;

    let branch_conditional : boolean = CurrentCond() != '111x';
    BranchTo{32}(new_pc, BranchType_ERET, branch_conditional);
    CheckExceptionCatch();
end;
```

G1.1.3.77 AArch32_ExecutingCP10or11Instr

```
// AArch32_ExecutingCP10or11Instr()
// =====

func AArch32_ExecutingCP10or11Instr() => boolean
begin
    let instr : bits(32) = ThisInstr();
    let instr_set : InstrSet = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
        return ((instr[27:24] == '1110' || instr[27:25] == '110') && instr[11:8] == '101x');
    else // InstrSet_T32
        return (instr[31:28] == '111x' && (instr[27:24] == '1110' || instr[27:25] == '110') &&
            instr[11:8] == '101x');
    end;
end;
end;
```

G1.1.3.78 AArch32_ITAdvance

```
// AArch32_ITAdvance()
// =====

func AArch32_ITAdvance()
begin
    if PSTATE.IT[2:0] == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT[4:0] = LSL(PSTATE.IT[4:0], 1);
    end;
    return;
end;
```

G1.1.3.79 AArch32_InterruptPending

```
// AArch32_InterruptPending()
// =====
// Returns TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise.

func AArch32_InterruptPending() => boolean
begin
    let pending_physical_interrupt : boolean = (IRQPending() || FIQPending() ||
        IsPhysicalSErrorPending());

    var pending_virtual_interrupt : boolean = FALSE;
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR().TGE == '0' then
        let virq_pending : boolean = (HCR().IMO == '1' && (VirtualIRQPending() || HCR().VI == '1'));
        let vfiq_pending : boolean = (HCR().FMO == '1' && (VirtualFIQPending() || HCR().VF == '1'));
        let vsei_pending : boolean = (HCR().AMO == '1' &&
            (IsVirtualSErrorPending() || HCR().VA == '1'));

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;
    end;

    return pending_physical_interrupt || pending_virtual_interrupt;
end;
```

G1.1.3.80 AArch32_SysRegRead

```
// AArch32_SysRegRead()
// =====
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].

impdef func AArch32_SysRegRead(cp_num : integer, instr : bits(32), t : integer)
```

```
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[21+:3];
    let CRn = instr[16+:4];
    let CRm = instr[0+:4];
    let opc2 = instr[5+:3];
end;
```

G1.1.3.81 AArch32_SysRegRead64

```
// AArch32_SysRegRead64()
// =====
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].

impdef func AArch32_SysRegRead64(cp_num : integer, instr : bits(32), t : integer, t2 : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[4+:4];
    let CRm = instr[0+:4];
end;
```

G1.1.3.82 AArch32_SysRegReadCanWriteAPSR

```
// AArch32_SysRegReadCanWriteAPSR()
// =====
// Determines whether the AArch32 System register read instruction can write to APSR flags.

func AArch32_SysRegReadCanWriteAPSR(cp_num : integer, instr : bits(32)) => boolean
begin
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr[11:8]);

    let opc1 : integer{} = UInt(instr[23:21]);
    let opc2 : integer{} = UInt(instr[7:5]);
    let CRn : integer{} = UInt(instr[19:16]);
    let CRm : integer{} = UInt(instr[3:0]);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;
    end;

    return FALSE;
end;
```

G1.1.3.83 AArch32_SysRegWrite

```
// AArch32_SysRegWrite()
// =====
// Read the contents of R[t] and write to a 32-bit AArch32 System register.

impdef func AArch32_SysRegWrite(cp_num : integer, instr : bits(32), t : integer)
begin
    let (-, el) = ELFromM32(PSTATE.M);
    let opc1 = instr[21+:3];
    let CRn = instr[16+:4];
    let CRm = instr[0+:4];
    let opc2 = instr[5+:3];
end;
```

G1.1.3.84 AArch32_SysRegWrite64

```
// AArch32_SysRegWrite64()
// =====
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.

impdef func AArch32_SysRegWrite64(cp_num : integer,
                                   instr : bits(32),
                                   t : integer,
                                   t2 : integer)

begin
    let (_, el) = ELFromM32(PSTATE.M);
    let opcl = instr[4+:4];
    let CRm = instr[0+:4];

end;
```

G1.1.3.85 AArch32_SysRegWriteM

```
// AArch32_SysRegWriteM()
// =====
// Read a value from a virtual address and write it to an AArch32 System register.

impdef func AArch32_SysRegWriteM(cp_num : integer, instr : bits(32), address : bits(32))

begin
    let CRd : bits(4) = instr[15:12];

end;
```

G1.1.3.86 AArch32_WriteMode

```
// AArch32_WriteMode()
// =====
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

func AArch32_WriteMode(mode : bits(5))

begin
    let (valid,el) : (boolean, bits(2)) = ELFromM32(mode);
    assert valid;
    PSTATE.M = mode;
    PSTATE.EL = el;
    PSTATE.nRW = '1';
    PSTATE.SP = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;

end;
```

G1.1.3.87 AArch32_WriteModeByInstr

```
// AArch32_WriteModeByInstr()
// =====
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

func AArch32_WriteModeByInstr(mode : bits(5))

begin
    var (valid,el) : (boolean, bits(2)) = ELFromM32(mode);

    // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    end;

    // * A change to or from Hyp mode.
```

```

    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;
    end;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32_WriteMode(mode);
    end;
end;
end;

```

G1.1.3.88 BadMode

```

// BadMode()
// =====

readonly func BadMode(mode : bits(5)) => boolean
begin
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    var valid : boolean;
    case mode of
        when M32_Monitor =>
            valid = HaveAArch32EL(EL3);
        when M32_Hyp =>
            valid = HaveAArch32EL(EL2);
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System =>
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32
            valid = HaveAArch32EL(EL1);
        when M32_User =>
            valid = HaveAArch32EL(EL0);
        otherwise =>
            valid = FALSE; // Passed an illegal mode value
    end;
    return !valid;
end;

```

G1.1.3.89 BankedRegisterAccessValid

```

// BankedRegisterAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

func BankedRegisterAccessValid(SYSm : bits(5), mode : bits(5)) => (boolean, Constraint)
begin
    var valid : boolean = TRUE;
    var c : Constraint = Constraint_NONE;
    var banked_unpred : boolean = FALSE;
    case SYSm of
        when '000xx', '00100' => // R8_usr to R12_usr
            banked_unpred = mode != M32_FIQ;
        when '00101' => // SP_usr
            banked_unpred = mode == M32_System;
        when '00110' => // LR_usr
            banked_unpred = mode IN {M32_Hyp, M32_System};
        when '010xx', '0110x', '01110' => // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            banked_unpred = mode == M32_FIQ;
        when '1000x' => // LR_irq, SP_irq
            banked_unpred = mode == M32_IRQ;
        when '1001x' => // LR_svc, SP_svc
            banked_unpred = mode == M32_Svc;
        when '1010x' => // LR_abt, SP_abt
    end;
    return (valid, c);
end;

```



```

        banked_unpred = mode == M32_Abort;
    when '1011x' =>                                     // LR_und, SP_und
        banked_unpred = mode == M32_Undef;
    when '11110' =>                                     // ELR_hyp, only from Monitor or Hyp mode
        banked_unpred = !HaveEL(EL2) || ! mode IN {M32_Monitor,M32_Hyp};
    when '11111' =>                                     // SP_hyp, only from Monitor mode
        banked_unpred = !HaveEL(EL2) || mode != M32_Monitor;
    otherwise =>
        valid = FALSE;
        c      = ConstrainUnpredictable(Unpredictable_UnimplementedRegister);
        assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_ANYREG};
end;

if banked_unpred then
    valid = FALSE;
    c      = ConstrainUnpredictable(Unpredictable_BankedRegister);
    assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_UNKNOWN};
end;

return (valid, c);
end;

```

G1.1.3.90 CPSRWriteByInstr

```

// CPSRWriteByInstr()
// =====
// Update PSTATE.[N,Z,C,V,Q,GR,E,A,I,F,M] from a CPSR value written by an MSR instruction.

func CPSRWriteByInstr(value : bits(32), bytemask : bits(4))
begin
    let privileged : boolean = PSTATE.EL != EL0;          // PSTATE.[A,I,F,M] are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask[3] == '1' then
        PSTATE.[N,Z,C,V,Q] = value[31:27];
        // Bits [26:24] are ignored
    end;

    if bytemask[2] == '1' then
        // Bit [23] is RES0
        // Bit [22] is RES0
        // Bit [21] is RES0
        // Bit [20] is RES0
        PSTATE.GE = value[19:16];
    end;

    if bytemask[1] == '1' then
        // Bits [15:10] are RES0
        PSTATE.E = value[9];                               // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value[8];
        end;
    end;

    if bytemask[0] == '1' then
        if privileged then
            PSTATE.[I,F] = value[7:6];
            // Bit [5] is RES0
            // AArch32_WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32_WriteModeByInstr(value[4:0]);
        end;
    end;

    return;
end;

```

G1.1.3.91 ConditionPassed

```
// ConditionPassed()
// =====

func ConditionPassed() => boolean
begin
    return ConditionHolds(CurrentCond());
end;
```

G1.1.3.92 CurrentCond

```
// CurrentCond()
// =====
// Returns the condition code that applies to the currently executing instruction

impdef func CurrentCond() => bits(4)
begin
    return Zeros{4};
end;
```

G1.1.3.93 InITBlock

```
// InITBlock()
// =====

func InITBlock() => boolean
begin
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT[3:0] != '0000';
    else
        return FALSE;
    end;
end;
```

G1.1.3.94 LastInITBlock

```
// LastInITBlock()
// =====

func LastInITBlock() => boolean
begin
    return (PSTATE.IT[3:0] == '1000');
end;
```

G1.1.3.95 SPSRWriteByInstr

```
// SPSRWriteByInstr()
// =====

func SPSRWriteByInstr(value : bits(32), bytemask : bits(4))
begin
    var new_spsr : bits(32) = SPSR_curr();

    if bytemask[3] == '1' then
        new_spsr[31:24] = value[31:24]; // N, Z, C, V, Q flags, IT[1:0], J bits
    end;

    if bytemask[2] == '1' then
        new_spsr[23:16] = value[23:16]; // IL bit, GE[3:0] flags
    end;

    if bytemask[1] == '1' then
        new_spsr[15:8] = value[15:8]; // IT[7:2] bits, E bit, A interrupt mask
    end;
end;
```

```

end;

if bytemask[0] == '1' then
    new_spsr[7:0] = value[7:0];      // I, F interrupt masks, T bit, Mode bits
end;

SPSR_curr() = new_spsr;            // UNPREDICTABLE if User or System mode

return;
end;

```

G1.1.3.96 SPSRaccessValid

```

// SPSRaccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

func SPSRaccessValid(SYSm : bits(5), mode : bits(5)) => (boolean, Constraint)
begin
    var valid : boolean = TRUE;
    var c : Constraint = Constraint_NONE;
    var banked_unpred : boolean = FALSE;
    case SYSm of
        when '01110' =>                                // SPSR_fiq
            banked_unpred = mode == M32_FIQ;
        when '10000' =>                                // SPSR_irq
            banked_unpred = mode == M32_IRQ;
        when '10010' =>                                // SPSR_svc
            banked_unpred = mode == M32_Svc;
        when '10100' =>                                // SPSR_abt
            banked_unpred = mode == M32_Abort;
        when '10110' =>                                // SPSR_und
            banked_unpred = mode == M32_Undef;
        when '11110' =>                                // SPSR_hyp
            banked_unpred = !HaveEL(EL2) || mode != M32_Monitor;
        otherwise =>
            valid = FALSE;
            c = ConstrainUnpredictable(Unpredictable_UnimplementedRegister);
            assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_ANYREG};
    end;

    if banked_unpred then
        valid = FALSE;
        c = ConstrainUnpredictable(Unpredictable_BankedRegister);
        assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_UNKNOWN};
    end;

    return (valid, c);
end;

```

G1.1.3.97 SelectInstrSet

```

// SelectInstrSet()
// =====

func SelectInstrSet(iset : InstrSet)
begin
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
end;

```

G1.1.3.98 AArch32_DTLBI_ALL

```
// AArch32_DTLBI_ALL()
// =====
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_DTLBI_ALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, attr_in : TLBMemAttr)

begin
    return;
end;
```

G1.1.3.99 AArch32_DTLBI_ASID

```
// AArch32_DTLBI_ASID()
// =====
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_DTLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, attr_in : TLBMemAttr, Rt : bits(32))

begin
    return;
end;
```

G1.1.3.100 AArch32_DTLBI_VA

```
// AArch32_DTLBI_VA()
// =====
// Invalidate by VA all stage 1 data TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_DTLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBMemAttr,
    Rt : bits(32))

begin
    return;
end;
```

G1.1.3.101 AArch32_EffectiveBroadcast

```
// AArch32_EffectiveBroadcast()
// =====

func AArch32_EffectiveBroadcast(broadcast_in : Broadcast) => Broadcast

begin
    var broadcast : Broadcast = broadcast_in;

    if PSTATE.EL != EL1 then return broadcast; end;

    if broadcast == Broadcast_NSH && HCR().FB == '1' then
        broadcast = Broadcast_ForcedISH;
    end;

    return broadcast;
end;
```

G1.1.3.102 AArch32_ITLBI_ALL

```
// AArch32_ITLBI_ALL()
// =====
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_ITLBI_ALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, attr_in : TLBIMemAttr)

begin
    return;
end;
```

G1.1.3.103 AArch32_ITLBI_ASID

```
// AArch32_ITLBI_ASID()
// =====
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID
// (where regime supports) and ASID in the parameter Rt in the indicated translation
// regime with the indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_ITLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, attr_in : TLBIMemAttr, Rt : bits(32))

begin
    return;
end;
```

G1.1.3.104 AArch32_ITLBI_VA

```
// AArch32_ITLBI_VA()
// =====
// Invalidate by VA all stage 1 instruction TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_ITLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Rt : bits(32))

begin
    return;
end;
```

G1.1.3.105 AArch32_TLBI_ALL

```
// AArch32_TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

func AArch32_TLBI_ALL(security : SecurityState, regime : Regime,
    broadcast : Broadcast, attr : TLBIMemAttr)

begin
    return;
end;
```

G1.1.3.106 AArch32_TLBI_ASID

```
// AArch32_TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_ASID(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, attr_in : TLBIMemAttr, Rt : bits(32))

begin
    return;
end;
```

G1.1.3.107 AArch32_TLBI_IPAS2

```
// AArch32_TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.

func AArch32_TLBI_IPAS2(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast : Broadcast, level : TLBILevel, attr : TLBIMemAttr, Rt : bits(32))

begin
    return;
end;
```

G1.1.3.108 AArch32_TLBI_VA

```
// AArch32_TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_VA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Rt : bits(32))

begin
    return;
end;
```

G1.1.3.109 AArch32_TLBI_VAA

```
// AArch32_TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

func AArch32_TLBI_VAA(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
    broadcast_in : Broadcast, level : TLBILevel, attr_in : TLBIMemAttr,
    Rt : bits(32))

begin
    return;
end;
```

G1.1.3.110 AArch32_TLBI_VMALL

```
// AArch32_TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

func AArch32_TLBI_VMALL(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                        broadcast_in : Broadcast, attr_in : TLBMemAttr)

begin
    return;
end;
```

G1.1.3.111 AArch32_TLBI_VMALLS12

```
// AArch32_TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// broadcast domain that match the indicated VMID.

func AArch32_TLBI_VMALLS12(security : SecurityState, regime : Regime, vmid : bits(NUM_VMIDBITS),
                           broadcast : Broadcast, attr : TLBMemAttr)

begin
    return;
end;
```

G1.1.3.112 Sat

```
// Sat()
// =====

func Sat{N}(i : integer, unsigned : boolean) => bits(N)
begin
    let result : bits(N) = if unsigned then UnsignedSat{N}(i) else SignedSat{N}(i);
    return result;
end;
```

G1.1.4 aarch32/translation

This section includes the following pseudocode functions:

- [AArch32_MAIRAttr](#)
- [AArch32_CheckBreakpoint](#)
- [AArch32_CheckDebug](#)
- [AArch32_CheckVectorCatch](#)
- [AArch32_CheckWatchpoint](#)
- [AArch32_S1CheckPermissions](#)
- [AArch32_S1HasAlignmentFaultDueToMemType](#)
- [AArch32_S2CheckPermissions](#)
- [AArch32_S2HasAlignmentFaultDueToMemType](#)
- [AArch32_AccessHasPermission](#)
- [AArch32_FullValidate](#)
- [AArch32_MPUValidate](#)
- [AArch32_S1BREnabled](#)

- [AArch32_S1Validate](#)
- [AArch32_S2Validate](#)
- [BackgroundMemoryAttr](#)
- [AArch32_EL2Enabled](#)
- [AArch32_S1DisabledOutput](#)
- [AArch32_S1Enabled](#)
- [AArch32_TranslateAddress](#)
- [AArch32_S1DCacheEnabled](#)
- [AArch32_S1ICacheEnabled](#)

G1.1.4.1 AArch32_MAIRAttr

```
// AArch32_MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

func AArch32_MAIRAttr(index : integer, mair : bits(64)) => bits(8)
begin
    assert (index < 8);
    return mair[index*8];
end;
```

G1.1.4.2 AArch32_CheckBreakpoint

```
// AArch32_CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

func AArch32_CheckBreakpoint(fault_in : FaultRecord, vaddress : bits(32),
                             accdesc : AccessDescriptor, size : integer) => FaultRecord
begin
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    var fault : FaultRecord = fault_in;
    var match : boolean = FALSE;
    var mismatch : boolean = FALSE;
    var brkptinfo : BreakpointInfo;

    for i = 0 to NumBreakpointsImplemented() - 1 do
        brkptinfo = AArch32_BreakpointMatch(i, vaddress, accdesc, size);
        match = match || brkptinfo.match;
        mismatch = mismatch || brkptinfo.mismatch;
    end;

    if match && HaltOnBreakpointOrWatchpoint() then
        let reason : bits(6) = DebugHalt_Breakpoint;
        Halt(reason);
    elseif (match || mismatch) then
        fault.statuscode = Fault_Debug;
        fault.debugmoe = DebugException_Breakpoint;
        fault.vaddress = ZeroExtend{64}(vaddress);
    end;

    return fault;
end;
```

G1.1.4.3 AArch32_CheckDebug

```
// AArch32_CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug state.

func AArch32_CheckDebug(vaddress : bits(32), accdesc : AccessDescriptor,
                        size : integer) => FaultRecord
begin

    var fault : FaultRecord = NoFault(accdesc, ZeroExtend{64}(vaddress));

    let d_side : boolean = IsWatchpointableAccess(accdesc);
    let i_side : boolean = (accdesc.acctype == AccessType_IFETCH);
    let generate_exception : boolean = (AArch32_GenerateDebugExceptions() &&
                                        DBGDSCRext().MDBGen == '1');
    let halt : boolean = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    let vector_catch_first : boolean = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
        fault = AArch32_CheckVectorCatch(fault, vaddress, size);
    end;

    if fault.statuscode == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32_CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch32_CheckBreakpoint(fault, vaddress, accdesc, size);
        end;
    end;

    if fault.statuscode == Fault_None && i_side && !vector_catch_first && generate_exception then
        return AArch32_CheckVectorCatch(fault, vaddress, size);
    end;

    return fault;
end;
```

G1.1.4.4 AArch32_CheckVectorCatch

```
// AArch32_CheckVectorCatch()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

func AArch32_CheckVectorCatch(fault_in : FaultRecord, vaddress : bits(32),
                              size : integer) => FaultRecord
begin
    assert ELUsingAArch32(S1TranslationRegime());

    var fault : FaultRecord = fault_in;
    var match : boolean = AArch32_VCRMATCH(vaddress);
    if size == 4 && !match && AArch32_VCRMATCH(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
    end;

    if match then
        fault.statuscode = Fault_Debug;
        fault.debugmoe = DebugException_VectorCatch;
    end;

    return fault;
end;
```

G1.1.4.5 AArch32_CheckWatchpoint

```
// AArch32_CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

func AArch32_CheckWatchpoint(fault_in : FaultRecord, vaddress : bits(32),
                             accdesc : AccessDescriptor, size : integer) => FaultRecord
begin
    assert ELUsingAArch32(S1TranslationRegime());
    var fault : FaultRecord = fault_in;
    var match : boolean = FALSE;

    if !IsWatchpointableAccess(accdesc) then
        return fault;
    end;

    for i = 0 to NumWatchpointsImplemented() - 1 do
        let watchptinfo : WatchpointInfo = AArch32_WatchpointMatch(i, vaddress, size, accdesc);
        match = match || watchptinfo.value_match;
    end;

    if match && HaltOnBreakpointOrWatchpoint() then
        let reason : bits(6) = DebugHalt_Watchpoint;
        EDWAR() = ZeroExtend{64}(vaddress);
        Halt(reason);
    elsif match then
        fault.statuscode = Fault_Debug;
        fault.debugmoe = DebugException_Watchpoint;
        fault.vaddress = ZeroExtend{64}(vaddress);
    end;

    return fault;
end;
```

G1.1.4.6 AArch32_S1CheckPermissions

```
// AArch32_S1CheckPermissions()
// =====
// Perform stage-1 permission check.

func AArch32_S1CheckPermissions(fault_in : FaultRecord, regime : Regime,
                                 memattrs : MemoryAttributes, permissions : Permissions,
                                 accdesc : AccessDescriptor) => FaultRecord
begin
    var r, w, x : bit;
    var fault : FaultRecord = fault_in;
    if HasUnprivileged(regime) then
        var pr, pw : bit;
        var ur, uw : bit;
        var ux, px : bit;
        // Apply MPU permissions given Translation Regime serves 2 ELs
        case permissions.ap[2:1] of
            when '00' => (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
            when '01' => (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' => (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
            when '11' => (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only
        end;

        x = NOT(permissions.xn);

        if SCTLRL().M == '1' then
            x = x AND NOT(SCTLRL().WXN AND (pw OR uw));
        end;
```

```

ux = x;
px = x AND NOT(SCTLR().UWXN AND uw);
(r,w,x) = if accdesc.el == ELO then (ur,uw,ux) else (pr,pw,px);
else
    // Apply MPU permissions given Translation Regime serves 1 EL
    case permissions.ap[2] of
        when '0' => (r,w) = ('1','1'); // No effect
        when '1' => (r,w) = ('1','0'); // Read-only
    end;
    x = NOT(permissions.xn);
    // HSCTLR.WXN further control the execute permission.
    if HSCTLR().M == '1' then
        x = x AND NOT(HSCTLR().WXN AND w);
    end;
end;

if accdesc.acctype == AccessType_IFETCH then
    if (ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT &&
        memattrs.memtype == MemType_Device) then
        fault.statuscode = Fault_Permission;

    elsif x == '0' || r == '0' then
        fault.statuscode = Fault_Permission;

    end;
elseif accdesc.acctype == AccessType_DC then
    if accdesc.cacheop == CacheOp_Invalidate then
        if w == '0' then
            fault.statuscode = Fault_Permission;
        end;

    else
        // DC ops from privileged context which clean cannot generate a Permission fault
        if accdesc.el == ELO && r == '0' then
            fault.statuscode = Fault_Permission;
        end;
    end;

elseif accdesc.acctype == AccessType_IC then
    // IC from privileged context cannot Permission fault
    if (accdesc.el == ELO && r == '0' &&
        ImpDefBool("Permission fault on ELO IC_IVAU execution")) then
        fault.statuscode = Fault_Permission;
    end;

elseif accdesc.read && r == '0' then
    fault.statuscode = Fault_Permission;
    fault.write      = FALSE;

elseif accdesc.write && w == '0' then
    fault.statuscode = Fault_Permission;
    fault.write      = TRUE;
end;

return fault;
end;

```

G1.1.4.7 AArch32_S1HasAlignmentFaultDueToMemType

```

// AArch32_S1HasAlignmentFaultDueToMemType()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses to memory type

func AArch32_S1HasAlignmentFaultDueToMemType(accdesc : AccessDescriptor, aligned : boolean,
    memattrs : MemoryAttributes) => boolean

```

```
begin
  if memattrs.memtype != MemType_Device then
    return FALSE;
  elsif accdesc.acctype == AccessType_DCZero then
    return TRUE;
  elsif !aligned then
    return !(ImpDefBool("Device location supports unaligned access"));
  else
    return FALSE;
  end;
end;
```

G1.1.4.8 AArch32_S2CheckPermissions

```
// AArch32_S2CheckPermissions()
// =====
// Perform stage-2 permission check.

func AArch32_S2CheckPermissions(fault_in : FaultRecord,
                                memattrs : MemoryAttributes,
                                permissions : Permissions,
                                accdesc : AccessDescriptor) => FaultRecord
begin
  // Stage 2 AP follows a similar mapping as stage 1
  // EL1 & EL0 are treated in the same way EL0 is in stage 1
  var r : bit;
  var w : bit;
  var fault : FaultRecord = fault_in;
  case permissions.ap[2:1] of
    when '00' => (r,w) = ('0','0'); // No access
    when '10' => (r,w) = ('0','0'); // No access
    when '01' => (r,w) = ('1','1'); // No effect
    when '11' => (r,w) = ('1','0'); // Read-only
  end;
  let x : bit = NOT(permissions.xn);

  if accdesc.acctype == AccessType_IFETCH then
    let constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    if constraint == Constraint_FAULT && memattrs.memtype == MemType_Device then
      fault.statuscode = Fault_Permission;

    elsif x == '0' || r == '0' then
      fault.statuscode = Fault_Permission;
    end;

  elsif accdesc.acctype == AccessType_DC then
    if accdesc.cacheop == CacheOp_Invalidate then
      if w == '0' then
        fault.statuscode = Fault_Permission;
      end;
    else
      // DC from privileged context which do no write cannot Permission fault
      if accdesc.el == EL0 && r == '0' then
        fault.statuscode = Fault_Permission;
      end;
    end;

  elsif accdesc.acctype == AccessType_IC then
    // IC from privileged context cannot Permission fault
    if (accdesc.el == EL0 && r == '0' &&
        ImpDefBool("Permission fault on EL0 IC_IVAU execution")) then
      fault.statuscode = Fault_Permission;
    end;
  elsif accdesc.read && r == '0' then
    fault.statuscode = Fault_Permission;
  end;
```

```

        fault.write      = FALSE;

    elsif accdesc.write && w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;
    end;

    return fault;
end;

```

G1.1.4.9 AArch32_S2HasAlignmentFaultDueToMemType

```

// AArch32_S2HasAlignmentFaultDueToMemType()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses due to memory type

func AArch32_S2HasAlignmentFaultDueToMemType(accdesc : AccessDescriptor, aligned : boolean,
                                              memattrs : MemoryAttributes) => boolean
begin
    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(ImpDefBool("Device location supports unaligned access"));
    else
        return FALSE;
    end;
end;

```

G1.1.4.10 AArch32_AccessHasPermission

```

// AArch32_AccessHasPermission()
// =====

func AArch32_AccessHasPermission(el : bits(2), regime : Regime) => boolean
begin
    return ((regime == Regime_EL2 && el == EL2) ||
            (regime == Regime_EL10 && el == EL1));
end;

```

G1.1.4.11 AArch32_FullValidate

```

// AArch32_FullValidate()
// =====
// Apply PMSA validation on memory access subject to
// configuration and translation regime

func AArch32_FullValidate(va : bits(32),
                          size : integer,
                          accdesc : AccessDescriptor,
                          aligned : boolean) => AddressDescriptor
begin
    var ipa : AddressDescriptor;
    var pa : AddressDescriptor;
    // Initialise fault record in case a fault is detected down the line
    let va_in : bits(64) = ZeroExtend{64}(va);
    var fault : FaultRecord = NoFault(accdesc, va_in);
    let regime : Regime = TranslationRegime(PSTATE.EL);

    // First Stage Validation
    (fault, ipa) = AArch32_S1Validate(fault, regime, va, aligned, accdesc);
    if fault.statuscode != Fault_None then

```

```

        return CreateFaultyAddressDescriptor(fault);
    end;

    // Second Stage Validation
    if regime == Regime_EL10 then
        (fault, pa) = AArch32_S2Validate(fault, ipa, aligned, accdesc);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(fault);
        end;
        return pa;
    else
        return ipa;
    end;
end;

```

G1.1.4.12 AArch32_MPUValidate

```

// AArch32_MPUValidate()
// =====
// Attempt to match the input address with an active Memory Protection Unit (MPU) and
// retrieve assigned permissions and memory attributes

func AArch32_MPUValidate(fault_in : FaultRecord,
    regime : Regime,
    is_stage2 : boolean,
    ia : bits(32)) => (FaultRecord, boolean, PRBARnType, PRLARnType)
begin
    var fault : FaultRecord = fault_in;
    var num_regions : integer;
    var matched_prbar : PRBARnType;
    var matched_prlar : PRLARnType;
    var matched : boolean = FALSE;
    if regime == Regime_EL2 then
        assert !is_stage2;
    end;

    case regime of
        when Regime_EL2 => num_regions = UInt(HMPUIR().REGION);
        when Regime_EL10 =>
            if is_stage2 then
                num_regions = UInt(HMPUIR().REGION);
            else
                num_regions = UInt(MPUIR().REGION);
            end;
    end;

    for index = 0 to num_regions-1 do
        var prbar : PRBARnType;
        var prlar : PRLARnType;
        case regime of
            when Regime_EL2 =>
                prbar = PRBAR_EL(EL2, index);
                prlar = PRLAR_EL(EL2, index);
            when Regime_EL10 =>
                if is_stage2 then
                    prbar = PRBAR_EL(EL2, index);
                    prlar = PRLAR_EL(EL2, index);
                else
                    prbar = PRBAR_EL(EL1, index);
                    prlar = PRLAR_EL(EL1, index);
                end;
        end;

        let baseaddr : bits(32) = ZeroExtend{32}(prbar.BASE :: Zeros{6});
        let limitaddr : bits(32) = ZeroExtend{32}(prlar.LIMIT :: Ones{6});
    end;
end;

```

```

// Check for a matching MPU region
if (prlar.EN == '1' &&
    UInt(ia) >= UInt(baseaddr) &&
    UInt(ia) <= UInt(limitaddr)) then

    // Check for multiple region match
    if matched then
        fault.statuscode = Fault_Translation;
        return (fault, matched, ARBITRARY : PRBARnType, ARBITRARY : PRLARnType);
    end;

    matched = TRUE;
    matched_prbar = prbar;
    matched_prlar = prlar;
end;

return (fault, matched, matched_prbar, matched_prlar);
end;

```

G1.1.4.13 AArch32_S1BREnabled

```

// AArch32_S1BREnabled()
// =====
// Determine the number of MPUs supported in PMSA stage 1 for given regime

func AArch32_S1BREnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime_EL2 =>
            return HSCTLR().BR == '1';
        when Regime_EL10 =>
            return HCR().[DC,TGE] == '00' && SCTLR().BR == '1';
    end;
end;

```

G1.1.4.14 AArch32_S1Validate

```

// AArch32_S1Validate()
// =====
// Perform stage 1 PMSA validation using Memory Protection Units (MPUs).

func AArch32_S1Validate(fault_in : FaultRecord, regime : Regime,
    va_in : bits(32), aligned : boolean,
    accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
begin
    var fault : FaultRecord = fault_in;
    let is_stage2 : boolean = FALSE;
    var matched : boolean = FALSE;
    var use_dc : boolean = FALSE;
    let va : bits(32) = va_in;
    // Prepare fault fields if one is detected
    fault.secondstage = is_stage2;
    fault.s2fslwalk = FALSE;
    fault.level = 0;

    var mpurecord : MPURecord;
    var s1_permissions : Permissions;
    var s1_attr : bits(8);
    var s1_sh : bits(2);

    if !AArch32_S1Enabled(regime) then
        if regime == Regime_EL10 && HCR().DC == '1' then
            use_dc = TRUE; // Use default cacheable settings

```

```

else
    (s1_attr, s1_sh, s1_permissions) = BackgroundMemoryAttr(va, accdesc, regime);
end;

else // MPU Enabled
    var prbar : PRBARnType;
    var prlar : PRLARnType;
    (fault, matched, prbar, prlar) = AArch32_MPUValidate(fault, regime, is_stage2, va);

    // More than one match - Translation fault
    if fault.statuscode != Fault_None then
        return (fault, ARBITRARY : AddressDescriptor);
    // Unique MPU match
    elsif matched then
        s1_permissions.ap[2:1] = prbar.AP;
        s1_permissions.xn      = prbar.XN;

        var s1_mair : bits(64);
        case regime of
            when Regime_EL2 => s1_mair = HMAIR1():HMAIR0();
            when Regime_EL10 => s1_mair = MAIR1():MAIR0();
        end;
        s1_attr = AArch32_MAIRAttr(UInt(prlar.AttrIndx), s1_mair);
        s1_sh   = prbar.SH;
    // No match
    else
        if AArch32_AccessHasPermission(accdesc.el, regime) && AArch32_S1BREnabled(regime) then
            if regime == Regime_EL10 && HCR().DC == '1' then
                use_dc = TRUE; // Use default cacheable settings
            else
                (s1_attr, s1_sh, s1_permissions) = BackgroundMemoryAttr(va, accdesc, regime);
            end;
        else
            fault.statuscode = Fault_Translation;
            return (fault, ARBITRARY : AddressDescriptor);
        end;
    end;
end;

if use_dc then
    mpurecord.memattrs.memtype      = MemType_Normal;
    mpurecord.memattrs.inner.attrs  = MemAttr_WB;
    mpurecord.memattrs.inner.hints  = MemHint_RWA;
    mpurecord.memattrs.inner.transient = FALSE;
    mpurecord.memattrs.outer.attrs  = MemAttr_WB;
    mpurecord.memattrs.outer.hints  = MemHint_RWA;
    mpurecord.memattrs.outer.transient = FALSE;
    mpurecord.memattrs.shareability = Shareability_NSH;

    mpurecord.permissions.ap        = '011'; // RW at any EL
    mpurecord.permissions.xn        = '0';

else
    mpurecord.memattrs      = S1DecodeMemAttrs(s1_attr, s1_sh, FALSE);
    mpurecord.permissions  = s1_permissions;
end;

if AArch32_S1HasAlignmentFaultDueToMemType(accdesc, aligned, mpurecord.memattrs) then
    fault.statuscode = Fault_Alignment;
    return (fault, ARBITRARY : AddressDescriptor);
end;

fault = AArch32_S1CheckPermissions(fault, regime, mpurecord.memattrs,
                                    mpurecord.permissions, accdesc);
if fault.statuscode != Fault_None then
    return (fault, ARBITRARY : AddressDescriptor);

```



```

end;

var memattrs : MemoryAttributes;
if ((accdesc.acctype == AccessType_IFETCH &&
    (mpurecord.memattrs.memtype == MemType_Device || !AArch32_S1ICacheEnabled(regime))) ||
    (accdesc.acctype != AccessType_IFETCH &&
    mpurecord.memattrs.memtype == MemType_Normal && !AArch32_S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
else
    memattrs = mpurecord.memattrs;
end;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && HCR().VM == '1' &&
    !ImpDefBool("Apply effective shareability at stage 1")) then
    memattrs.shareability = mpurecord.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);
end;

let va_addr = ZeroExtend{64}(va);
// Output Address
var oa : FullAddress;
oa.address = va_addr[NUM_PABITS-1:0];
oa.paspace = PAS_NonSecure;
let ipa = CreateAddressDescriptor(va_addr, oa, memattrs, accdesc);
return (fault, ipa);
end;

```

G1.1.4.15 AArch32_S2Validate

```

// AArch32_S2Validate()
// =====
// Perform stage 2 FMSA validation using Memory Protection Units (MPUs).

func AArch32_S2Validate(fault_in : FaultRecord, ipa : AddressDescriptor,
    aligned : boolean,
    accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
begin
    var fault : FaultRecord = fault_in;
    let is_stage2 : boolean = TRUE;
    var matched : boolean = FALSE;
    // Prepare fault fields in case a fault is detected
    fault.statuscode = Fault_None; // Ignore any faults from stage 1
    fault.secondstage = is_stage2;
    fault.s2fslwalk = FALSE;
    fault.level = 0;
    fault.ipaddress = ipa.paddress.address;

    var s2_permissions : Permissions;
    var s2_attr : bits(8);
    var s2_sh : bits(2);
    var pa : AddressDescriptor;
    let addr : bits(32) = ipa.paddress.address[31:0];

    // Stage 2 is disabled
    if HCR().[DC,VM] == '00' then
        pa = ipa;
        return (fault, pa);
    end;

    if HSCTLR().M == '0' then
        (s2_attr, s2_sh, s2_permissions) = BackgroundMemoryAttr(addr, accdesc, Regime_EL10);
    else

```

```

var prbar : PRBARnType;
var prlar : PRLARnType;

(fault, matched, prbar, prlar) = AArch32_MPUValidate(fault, Regime_EL10, is_stage2, addr);
// More than one match - Translation fault
if fault.statuscode != Fault_None then
    return (fault, ARBITRARY : AddressDescriptor);
// Unique MPU match
elseif matched then
    s2_permissions.ap[2:1] = prbar.AP;
    s2_permissions.xn      = prbar.XN;

    var s2_mair : bits(64);
    s2_mair = HMAIR1():HMAIR0();
    s2_attr = AArch32_MAIRAttr(UInt(prlar.AttrIndx), s2_mair);
    s2_sh   = prbar.SH;

// No match
else
    fault.statuscode = Fault_Translation;
    return (fault, ARBITRARY : AddressDescriptor);
end;

end;

var mpurecord : MPUREcord;
mpurecord.permissions = s2_permissions;

// stage 2 memory attributes are decoded in the same way as stage 1
// for the EL1&0 regime. The only difference is allocation hints are ignored
mpurecord.memattrs = S1DecodeMemAttrs(s2_attr, s2_sh, FALSE);

if AArch32_S2HasAlignmentFaultDueToMemType(accdesc, aligned, mpurecord.memattrs) then
    fault.statuscode = Fault_Alignment;
    return (fault, ARBITRARY : AddressDescriptor);
end;

if (VSCTLR().S2DMA == '1' && mpurecord.memattrs.memtype == MemType_Device &&
    accdesc.a32lsmd && !accdesc.within8byte) then
    fault.statuscode = Fault_Permission;
    return (fault, ARBITRARY : AddressDescriptor);
end;

fault = AArch32_S2CheckPermissions(fault, mpurecord.memattrs, mpurecord.permissions, accdesc);

if fault.statuscode != Fault_None then
    return (fault, ARBITRARY : AddressDescriptor);
end;

var s2_memattrs : MemoryAttributes;

if ((accdesc.acctype == AccessType_IFETCH &&
    (mpurecord.memattrs.memtype == MemType_Device)) ||
    (accdesc.acctype != AccessType_IFETCH &&
    mpurecord.memattrs.memtype == MemType_Normal)) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = NormalNCMemAttr();
else
    s2_memattrs = mpurecord.memattrs;
end;

let memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, FALSE);

// Output Address
var oa : FullAddress;
oa.address = ipa.paddress.address;
oa.paspace = PAS_NonSecure;

```

```

    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs, accdesc);
    return (fault, pa);
end;

```

G1.1.4.16 BackgroundMemoryAttr

```

// BackgroundMemoryAttr()
// =====
// Background memory region attributes and permissions for a given address for data access.

func BackgroundMemoryAttr(address : bits(32) ,
                           accdesc : AccessDescriptor,
                           regime : Regime) => (bits(8), bits(2), Permissions)

begin
    var attr : bits(8);
    var sh : bits(2);
    var xn : bit;
    var ap : bits(3);
    var cacheable : boolean;

    if accdesc.acctype == AccessType_IFETCH then
        cacheable = if regime == Regime_EL2 then HSCTLR().I == '1' else SCTLR().I == '1';
        if UInt(address) <= 0x7FFFFFFF then
            if cacheable then
                attr = '10101010'; // Normal, iWT-rAllocate, oWT-rAllocate, Non-transient
                sh = '00'; // nonSH
                xn = '0';
                ap = '011'; // Full access
            else
                attr = '01000100'; // Normal, NC
                sh = '10'; // oSH
                xn = '0';
                ap = '011'; // Full access
            end;
        else
            xn = '1'; // Never Execute block
            attr = ARBITRARY : bits(8);
            sh = ARBITRARY : bits(2);
            ap = ARBITRARY : bits(3);
        end;
    else
        cacheable = if regime == Regime_EL2 then HSCTLR().C == '1' else SCTLR().C == '1';
        if UInt(address) >= 0x00000000 && UInt(address) <= 0x3FFFFFFF then
            if cacheable then
                attr = '11111111'; // Normal iWB-rwAllocate, oWB-rwAllocate, non-transient
                sh = '00'; // nonSH
                xn = '0';
                ap = '011'; // Full access
            else
                attr = '01000100'; // Normal NC
                sh = '10'; // oSH
                xn = '0';
                ap = '011'; // Full access
            end;
        elseif UInt(address) >= 0x40000000 && UInt(address) <= 0x5FFFFFFF then
            if cacheable then
                attr = '10111011'; // Normal iWT-rwAllocate, oWT-rwAllocate, non-transient
                sh = '00'; // nonSH
                xn = '0';
                ap = '011'; // Full access
            else
                attr = '01000100'; // Normal NC
                sh = '10'; // oSH
                xn = '0';
                ap = '011'; // Full access
            end;
        else
            attr = '00000000'; // Invalid
            sh = '00';
            xn = '1';
            ap = '000'; // Invalid
        end;
    end;
end;

```

```

        end;
    elsif UInt(address) >= 0x60000000 && UInt(address) <= 0x7FFFFFFF then
        attr = '01000100';    // Normal NC
        sh   = '10';          // oSH
        xn   = '0';
        ap   = '011';          // Full access
    elsif UInt(address) >= 0x80000000 && UInt(address) <= 0xBFFFFFFF then
        attr = '00000100';    // Device nGnRE
        sh   = '10';          // oSH
        xn   = '1';
        ap   = '011';          // Full access
    elsif UInt(address) >= 0xC0000000 && UInt(address) <= 0xFFFFFFFF then
        attr = '00000000';    // Device nGnRnE
        sh   = '10';          // oSH
        xn   = '1';
        ap   = '011';          // Full access
    end;
end;

var permissions : Permissions;
permissions.ap = ap;
permissions.xn = xn;
return (attr, sh, permissions);
end;

```

G1.1.4.17 AArch32_EL2Enabled

```

// AArch32_EL2Enabled()
// =====
// Returns whether EL2 is enabled for the given Security State

func AArch32_EL2Enabled(ss : SecurityState) => boolean
begin
    return ss == SS_NonSecure && HaveEL(EL2);
end;

```

G1.1.4.18 AArch32_S1DisabledOutput

```

// AArch32_S1DisabledOutput()
// =====
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

func AArch32_S1DisabledOutput(fault_in : FaultRecord, regime : Regime,
                               va : bits(32), aligned : boolean,
                               accdesc : AccessDescriptor) => (FaultRecord, AddressDescriptor)
begin

    var fault : FaultRecord = fault_in;
    var memattrs : MemoryAttributes;
    var default_cacheable : bit;
    default_cacheable = HCR().DC;

    if default_cacheable == '1' then
        // Use default cacheable settings
        memattrs.memtype = MemType_Normal;
        memattrs.inner.attrs = MemAttr_WB;
        memattrs.inner.hints = MemHint_RWA;
        memattrs.outer.attrs = MemAttr_WB;
        memattrs.outer.hints = MemHint_RWA;
        memattrs.shareability = Shareability_NSH;
    elsif accdesc.acctype == AccessType_IFETCH then
        memattrs.memtype = MemType_Normal;
        memattrs.shareability = Shareability_OSH;
        if AArch32_S1ICacheEnabled(regime) then
            memattrs.inner.attrs = MemAttr_WT;

```

```

        memattrs.inner.hints = MemHint_RA;
        memattrs.outer.attrs = MemAttr_WT;
        memattrs.outer.hints = MemHint_RA;
    else
        memattrs.inner.attrs = MemAttr_NC;
        memattrs.outer.attrs = MemAttr_NC;
    end;
else
    // Treat memory region as Device
    memattrs.memtype      = MemType_Device;
    memattrs.device       = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
end;

if AArch32_S1HasAlignmentFaultDueToMemType(accdesc, aligned, memattrs) then
    fault.statuscode = Fault_Alignment;
    return (fault, ARBITRARY : AddressDescriptor);
elsif (accdesc.exclusive &&
        !(regime == Regime_EL10 && EL2Enabled() && HCR().DC == '1') &&
        ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
    fault.statuscode = Fault_Exclusive;
end;

var oa : FullAddress;
oa.address = ZeroExtend{NUM_PABITS}(va);
oa.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
let ipa : AddressDescriptor = CreateAddressDescriptor(ZeroExtend{64}(va),
                                                    oa,
                                                    memattrs,
                                                    accdesc);

return (fault, ipa);
end;

```

G1.1.4.19 AArch32_S1Enabled

```

// AArch32_S1Enabled()
// =====
// Returns whether stage 1 translation is enabled for the active translation regime

func AArch32_S1Enabled(regime : Regime) => boolean
begin
    if regime == Regime_EL2 then
        return HSCTLR().M == '1';
    else
        return EL2Enabled() && HCR().TGE == '0' && SCTLR().M == '1';
    end;
end;

```

G1.1.4.20 AArch32_TranslateAddress

```

// AArch32_TranslateAddress()
// =====
// Main entry point for translating an address

func AArch32_TranslateAddress(va : bits(32), accdesc : AccessDescriptor,
                              aligned : boolean, size : integer) => AddressDescriptor
begin
    let regime : Regime = TranslationRegime(PSTATE.EL);

    var result : AddressDescriptor = AArch32_FullValidate(va, size, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType_IFETCH then
        // For an instruction fetch, CheckDebug will be called
    end;
end;

```

```
        // after the instruction is read from memory
        result.fault = AArch32_CheckDebug(va, accdesc, size);
    end;

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(64)(va);

    return result;
end;
```

G1.1.4.21 AArch32_S1DCacheEnabled

```
// AArch32_S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

func AArch32_S1DCacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime_EL2 => return HSCTLR().C == '1';
        when Regime_EL10 =>
            return SCTLRL().C == '1';
    end;
end;
```

G1.1.4.22 AArch32_S1ICacheEnabled

```
// AArch32_S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

func AArch32_S1ICacheEnabled(regime : Regime) => boolean
begin
    case regime of
        when Regime_EL2 => return HSCTLR().I == '1';
        when Regime_EL10 =>
            return SCTLRL().I == '1';
    end;
end;
```

G1.2 Shared pseudocode

This section holds the pseudocode that is common to execution in AArch64 state and in AArch32 state. Functions listed in this section are identified only by a `FunctionName`, without an `AArch32.` prefix. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `shared/debug/DebugTarget`.

The top-level sections of the shared pseudocode hierarchy are:

- [shared/debug](#).
- [shared/exceptions](#).
- [shared/functions](#).
- [shared/trace](#).
- [shared/translation](#).

G1.2.1 shared/debug

This section includes the following pseudocode functions:

- [ClearStickyErrors](#)
- [DebugTarget](#)
- [DebugTargetFrom](#)
- [DoubleLockStatus](#)
- [OSLockStatus](#)
- [Component](#)
- [GetAccessComponent](#)
- [SoftwareLockStatus](#)
- [IsGIAActivityMonitorImplemented](#)
- [IsGIAActivityMonitorOffsetImplemented](#)
- [AllowExternalDebugAccess](#)
- [AllowExternalPMUAccess](#)
- [Debug](#)
- [ExternalInvasiveDebugEnabled](#)
- [ExternalNoninvasiveDebugAllowed](#)
- [ExternalNoninvasiveDebugEnabled](#)
- [ExternalSecureInvasiveDebugEnabled](#)
- [ExternalSecureNoninvasiveDebugEnabled](#)
- [IsAccessNonSecure](#)
- [IsAccessSecure](#)
- [IsCorePowered](#)
- [BreakpointInfo](#)
- [CheckValidStateMatch](#)
- [ContextAwareBreakpointRange](#)
- [IsContextAwareBreakpoint](#)
- [NumBreakpointsImplemented](#)
- [NumContextAwareBreakpointsImplemented](#)
- [NumWatchpointsImplemented](#)
- [CTI_ProcessEvent](#)
- [CTI_SetEventLevel](#)
- [CTI_SignalEvent](#)

- [CrossTrigger](#)
- [CheckForDCCInterrupts](#)
- [DTR](#)
- [Read_DBGDTRRX_EL0](#)
- [Read_DBGDTRTX_EL0](#)
- [Read_DBGDTR_EL0](#)
- [Write_DBGDTRRX_EL0](#)
- [Write_DBGDTRTX_EL0](#)
- [Write_DBGDTR_EL0](#)
- [Write_EDITR](#)
- [DCPSInstruction](#)
- [DRPSInstruction](#)
- [DebugHalt](#)
- [DebugRestorePSR](#)
- [DisableITRAndResumeInstructionPrefetch](#)
- [ExecuteA64](#)
- [ExecuteT32](#)
- [ExitDebugState](#)
- [Halt](#)
- [HaltOnBreakpointOrWatchpoint](#)
- [Halted](#)
- [HaltingAllowed](#)
- [Restarting](#)
- [StopInstructionPrefetchAndEnableITR](#)
- [UpdateDbgAuthStatus](#)
- [UpdateEDSCRFIELDS](#)
- [haltingevents](#)
- [CheckEDBGRQ](#)
- [CheckExceptionCatch](#)
- [CheckExternalDebugRequestEvents](#)
- [CheckHaltingStep](#)
- [CheckOSUnlockCatch](#)
- [CheckPendingOSUnlockCatch](#)
- [CheckPendingResetCatch](#)
- [CheckResetCatch](#)
- [CheckSoftwareAccessToDebugRegisters](#)
- [ExternalDebugRequest](#)
- [HSAdvance](#)
- [ExternalDebugInterruptsDisabled](#)
- [pmu](#)
- [ApplyPMUZeroMask](#)
- [CheckForPMUOverflow](#)
- [CheckPMUOverflowCondition](#)
- [CountPMUEvents](#)
- [EffectiveHPMN](#)
- [ExtRegGetPMUAccessMask](#)
- [ExtRegZeroCycleCounter](#)

- [ExtRegZeroEventCounters](#)
- [ExtRegZeroPMUCounters](#)
- [GetNumEventCountersAccessible](#)
- [GetNumEventCountersSelfHosted](#)
- [GetPMUCounterRange](#)
- [GetPMUReadMask](#)
- [GetPMUWriteMask](#)
- [HasElapsed64Cycles](#)
- [PMUCounterRange](#)
- [PMUEvent](#)
- [PMUOverflowCondition](#)
- [PMUSwIncrement](#)
- [SysRegGetPMUAccessMask](#)
- [SysRegZeroCycleCounter](#)
- [SysRegZeroEventCounters](#)
- [SysRegZeroPMUCounters](#)
- [CreatePCSample](#)
- [PCSample](#)
- [Read_EDPCSRlo](#)
- [DebugExceptionReturnSS](#)
- [DataCacheWatchpointSize](#)
- [WatchpointInfo](#)

G1.2.1.1 ClearStickyErrors

```
// ClearStickyErrors()
// =====

func ClearStickyErrors()
begin
    EDSCR().TXU = '0';           // Clear TX underrun flag
    EDSCR().RXO = '0';           // Clear RX overrun flag

    if Halted() then             // in Debug state
        EDSCR().ITO = '0';       // Clear ITR overrun flag
    end;

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if (Halted() && EDSCR().ITE == '0' &&
        ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO)) then
        return;
    end;
    EDSCR().ERR = '0';           // Clear cumulative error flag

    return;
end;
```

G1.2.1.2 DebugTarget

```
// DebugTarget()
// =====
// Returns the debug exception target Exception level

func DebugTarget() => bits(2)
begin
```

```
    let ss : SecurityState = CurrentSecurityState();  
    return DebugTargetFrom(ss);  
end;
```

G1.2.1.3 DebugTargetFrom

```
// DebugTargetFrom()  
// =====  
  
func DebugTargetFrom(from_state : SecurityState) => bits(2)  
begin  
    var route_to_el2 : boolean;  
    if HaveEL(EL2) && from_state != SS_Secure then  
        route_to_el2 = (HDCR().TDE == '1' || HCR().TGE == '1');  
    else  
        route_to_el2 = FALSE;  
    end;  
  
    var target : bits(2);  
    if route_to_el2 then  
        target = EL2;  
    elseif HaveEL(EL3) && !HaveAArch64() && from_state == SS_Secure then  
        target = EL3;  
    else  
        target = EL1;  
    end;  
  
    return target;  
end;
```

G1.2.1.4 DoubleLockStatus

```
// DoubleLockStatus()  
// =====  
// Returns the state of the OS Double Lock.  
// FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.  
// TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.  
  
func DoubleLockStatus() => boolean  
begin  
    return DBGOSDLR().DLK == '1' && DBGPRCR().CORENPDRQ == '0' && !Halted();  
end;
```

G1.2.1.5 OSLockStatus

```
// OSLockStatus()  
// =====  
// Returns the state of the OS Lock.  
  
readonly func OSLockStatus() => boolean  
begin  
    return DBGOSLSR().OSLK == '1';  
end;
```

G1.2.1.6 Component

```
// Component  
// =====  
// Component Types.  
  
type Component of enumeration {  
    Component_GIC,  
    Component_PMU,  
    Component_Debug,
```

```

        Component_CTI
    };

```

G1.2.1.7 GetAccessComponent

```

// GetAccessComponent()
// =====
// Determines the component by decoding the physical address in the address descriptor.

impdef func GetAccessComponent(addrdesc : AddressDescriptor) => Component
begin
    return Component_Debug;
end;

```

G1.2.1.8 SoftwareLockStatus

```

// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

func SoftwareLockStatus(addrdesc : AddressDescriptor) => boolean
begin
    let component : Component = GetAccessComponent(addrdesc);
    if !HaveSoftwareLock(component) then
        return FALSE;
    end;
    case component of
        when Component_Debug => return EDLSR().SLK == '1';
        when Component_PMU   => return PMLSR().SLK == '1';
        when Component_CTI   => return CTILSR().SLK == '1';
        otherwise             => return FALSE;
    end;
end;

```

G1.2.1.9 IsG1ActivityMonitorImplemented

```

// IsG1ActivityMonitorImplemented()
// =====
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.

impdef func IsG1ActivityMonitorImplemented(i : integer) => boolean
begin
    return FALSE;
end;

```

G1.2.1.10 IsG1ActivityMonitorOffsetImplemented

```

// IsG1ActivityMonitorOffsetImplemented()
// =====
// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.

func IsG1ActivityMonitorOffsetImplemented(i : integer) => boolean
begin
    return FALSE;
end;

```

G1.2.1.11 AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External DBGEBVR[n]_EL1,
// DBGBCR[n]_EL1, DBGWVR[n]_EL1, DBGWCR[n]_EL1 registers, and, from Armv8.2, the
// OSLAR_EL1 register is allowed for the access. Returns FALSE otherwise.

func AllowExternalDebugAccess(addrdesc : AddressDescriptor) => boolean
begin
    // The access may also be subject to OS Lock, power-down, etc.
    if !ExternalInvasiveDebugEnabled() then return FALSE; end;
    if ExternalSecureInvasiveDebugEnabled() then return TRUE; end;
    return NonSecureOnlyImplementation();
end;
```

G1.2.1.12 AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.

func AllowExternalPMUAccess(addrdesc : AddressDescriptor) => boolean
begin
    // The access may also be subject to OS Lock, power-down, etc.
    if !ExternalInvasiveDebugEnabled() then return FALSE; end;
    if ExternalSecureInvasiveDebugEnabled() then return TRUE; end;
    if HDCR().EPMAD == '0' then
        return TRUE;
    else
        return ExternalHypNoninvasiveDebugEnabled();
    end;
end;
```

G1.2.1.13 Debug

```
// Debug authentication signals
// =====

var DBGEN : Signal;
var NIDEN : Signal;
var SPIDEN : Signal;
var SPNIDEN : Signal;
var HIDEN : Signal;
var HNIDEN : Signal;
```

G1.2.1.14 ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

readonly func ExternalInvasiveDebugEnabled() => boolean
begin
    return DBGEN == HIGH;
end;
```

G1.2.1.15 ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =====
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

func ExternalNoninvasiveDebugAllowed() => boolean
begin
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);
end;

// ExternalNoninvasiveDebugAllowed()
// =====

func ExternalNoninvasiveDebugAllowed(el : bits(2)) => boolean
begin
    if !ExternalNoninvasiveDebugEnabled() then return FALSE; end;
    if HaveEL(EL2) && (el == EL2 || HCR().TGE == '1') then
        return ExternalHypNoninvasiveDebugEnabled();
    end;
    return TRUE;
end;
```

G1.2.1.16 ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

readonly func ExternalNoninvasiveDebugEnabled() => boolean
begin
    return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
end;
```

G1.2.1.17 ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

readonly func ExternalSecureInvasiveDebugEnabled() => boolean
begin
    if !HaveSecureState() then return FALSE; end;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
end;
```

G1.2.1.18 ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

readonly func ExternalSecureNoninvasiveDebugEnabled() => boolean
begin
    if !HaveSecureState() then return FALSE; end;
    return (ExternalNoninvasiveDebugEnabled() &&
        (SPIDEN == HIGH || SPNIDEN == HIGH));
end;
```

G1.2.1.19 IsAccessNonSecure

```
// IsAccessNonSecure()
// =====
// Returns TRUE when an access is Non-Secure

func IsAccessNonSecure(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.paddress.paspace == PAS_NonSecure;
end;
```

G1.2.1.20 IsAccessSecure

```
// IsAccessSecure()
// =====
// Returns TRUE when an access is Secure

func IsAccessSecure(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.paddress.paspace == PAS_Secure;
end;
```

G1.2.1.21 IsCorePowered

```
// IsCorePowered()
// =====
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.

impdef func IsCorePowered() => boolean
begin
    return TRUE;
end;
```

G1.2.1.22 BreakpointInfo

```
// BreakpointInfo
// =====
// Breakpoint related fields.

type BreakpointInfo of record {
    match : boolean,          // breakpoint match
    mismatch : boolean        // breakpoint mismatch
};
```

G1.2.1.23 CheckValidStateMatch

```
// CheckValidStateMatch()
// =====
// Checks for an invalid state match that will generate Constrained
// Unpredictable behavior, otherwise returns Constraint_NONE.

func CheckValidStateMatch(ssc : bits(2), hmc : bit, pxc : bits(2),
    isbreakpnt : boolean) => (Constraint, bits(2), bit, bits(2))
begin
    var reserved : boolean = FALSE;

    // Values that are not allocated in any architecture version
    if (hmc::ssc::pxc) IN {'0 11 10', '1 00 x0', '1 01 10', '1 1x 10'} then
        reserved = TRUE;
    end;

    // Reserved when HMC is 1 or SSC is not 0b00, when both EL3 and EL2 are not implemented.
    if !HaveEL(EL3) && !HaveEL(EL2) && (hmc == '1' || ssc != '00') then
```

```

        reserved = TRUE;
    end;

    // Reserved when HMC is 1, SSC is 0b11, and PxC is 0b00, when EL2 is not implemented.
    if !HaveEL(EL2) && hmc::ssc::pxc == '1 11 00' then
        reserved = TRUE;
    end;

    // 'Usr/Sys/Svc' encoding is valid only for AArch32 EL1 breakpoints.
    // Otherwise reserved.
    if (!HaveAArch32EL(EL1) || !isbreakpnt) && hmc::pxc == '000' && ssc != '11' then
        reserved = TRUE;
    end;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        var c : Constraint;
        var unpred_state_bits : bits(5);
        var unpred_hmc : bit;
        var unpred_ssc : bits(2);
        var unpred_pxc : bits(2);

        (c, unpred_state_bits) = ConstrainUnpredictableBits{5}(Unpredictable_RESBPWPCTRL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};

        if c == Constraint_DISABLED then
            unpred_hmc = ARBITRARY : bit;
            unpred_ssc = ARBITRARY : bits(2);
            unpred_pxc = ARBITRARY : bits(2);
            return (c, unpred_ssc, unpred_hmc, unpred_pxc);
        else
            unpred_hmc = unpred_state_bits[4];
            unpred_ssc = unpred_state_bits[3:2];
            unpred_pxc = unpred_state_bits[1:0];
            return (Constraint_NONE, unpred_ssc, unpred_hmc, unpred_pxc);
        end;
    end;

    return (Constraint_NONE, ssc, hmc, pxc);
end;

```

G1.2.1.24 ContextAwareBreakpointRange

```

// ContextAwareBreakpointRange()
// =====
// Returns two numbers indicating the index of the first and last context-aware breakpoint.

func ContextAwareBreakpointRange() => (integer, integer)
begin
    let b : integer = NumBreakpointsImplemented();
    let c : integer = NumContextAwareBreakpointsImplemented();

    return (b - c, b - 1);
end;

```

G1.2.1.25 IsContextAwareBreakpoint

```

// IsContextAwareBreakpoint()
// =====
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

func IsContextAwareBreakpoint(n : integer) => boolean
begin
    let (lower, upper) : (integer, integer) = ContextAwareBreakpointRange();

```

```
        return n >= lower && n <= upper;
    end;
```

G1.2.1.26 NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// =====
// Returns the number of breakpoints implemented.

readonly func NumBreakpointsImplemented() => integer
begin
    return ImpDefInt("Number of breakpoints");
end;
```

G1.2.1.27 NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =====
// Returns the number of context-aware breakpoints implemented.

readonly func NumContextAwareBreakpointsImplemented() => integer
begin
    return ImpDefInt("Number of context-aware breakpoints");
end;
```

G1.2.1.28 NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// =====
// Returns the number of watchpoints implemented.

readonly func NumWatchpointsImplemented() => integer
begin
    return ImpDefInt("Number of watchpoints");
end;
```

G1.2.1.29 CTI_ProcessEvent

```
// CTI_ProcessEvent()
// =====
// Process a discrete event on a Cross Trigger output event trigger.

impdef func CTI_ProcessEvent(id : CrossTriggerOut)
begin
    return;
end;
```

G1.2.1.30 CTI_SetEventLevel

```
// CTI_SetEventLevel()
// =====
// Set a Cross Trigger multi-cycle input event trigger to the specified level.

impdef func CTI_SetEventLevel(id : CrossTriggerIn, level : Signal)
begin
    return;
end;
```

G1.2.1.31 CTI_SignalEvent

```
// CTI_SignalEvent()
// =====
// Signal a discrete event on a Cross Trigger input event trigger.

impdef func CTI_SignalEvent(id : CrossTriggerIn)
begin
    return;
end;
```

G1.2.1.32 CrossTrigger

```
// CrossTrigger
// =====

type CrossTriggerOut of enumeration {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                                     CrossTriggerOut_IRQ,           CrossTriggerOut_RSVD3,
                                     CrossTriggerOut_TraceExtIn0,    CrossTriggerOut_TraceExtIn1,
                                     CrossTriggerOut_TraceExtIn2,    CrossTriggerOut_TraceExtIn3};

type CrossTriggerIn of enumeration {CrossTriggerIn_CrossHalt,      CrossTriggerIn_PMUOverflow,
                                    CrossTriggerIn_SPESample,      CrossTriggerIn_RSVD3,
                                    CrossTriggerIn_TraceExtOut0,    CrossTriggerIn_TraceExtOut1,
                                    CrossTriggerIn_TraceExtOut2,    CrossTriggerIn_TraceExtOut3,
                                    CrossTriggerIn_TRBESStop,       CrossTriggerIn_TRBEMgmt,
                                    CrossTriggerIn_TRBEWrap};
```

G1.2.1.33 CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =====

func CheckForDCCInterrupts()
begin
    let commrx : boolean = (EDSCR().RXfull == '1');
    let commtx : boolean = (EDSCR().TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    var commirq : boolean;
    commirq = ((commrx && DBGDCCINT().RX == '1') ||
              (commtx && DBGDCCINT().TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
end;
```

G1.2.1.34 DTR

```
// DTR
// ===

var DTRRX : bits(32);
var DTRTX : bits(32);
```

G1.2.1.35 Read_DBGDTRRX_EL0

```
// Read_DBGDTRRX_EL0()
// =====
// Called on reads of debug register 0x080.

func Read_DBGDTRRX_EL0(memory_mapped : boolean) => bits(32)
begin
    return DTRRX;
end;
```

G1.2.1.36 Read_DBGDTRTX_EL0

```
// Read_DBGDTRTX_EL0()
// =====
// Called on reads of debug register 0x08C.

func Read_DBGDTRTX_EL0(memory_mapped : boolean) => bits(32)
begin
    let underrun : boolean = (EDSCR().TXfull == '0' ||
                              (Halted() && EDSCR().MA == '1' && EDSCR().ITE == '0'));
    let value : bits(32) = if underrun then ARBITRARY : bits(32) else DTRTX;

    if EDSCR().ERR == '1' then return value; end;           // Error flag set: no side-effects

    if underrun then
        EDSCR().TXU = '1'; EDSCR().ERR = '1';              // Underrun condition: block side-effects
        return value;                                       // Return UNKNOWN
    end;

    EDSCR().TXfull = '0';
    if Halted() && EDSCR().MA == '1' then
        EDSCR().ITE = '0';                                // See comments in Write_EDITR()

        ExecuteT32(0xF850[15:0] /*hw1*/, 0x1B04[15:0] /*hw2*/); // T32 "LDR R1, [R0], #4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR().ERR == '1' then
            EDSCR().TXfull = ARBITRARY : bit;
        else
            ExecuteT32(0xEE00[15:0] /*hw1*/, 0x1E15[15:0] /*hw2*/); // T32 "MSR DBGDTRTXint, R1"
            // "MSR DBGDTRTX_EL0, X1" calls Write_DBGDTR_EL0() which sets TXfull.
            assert EDSCR().TXfull == '1';
        end;
        R(1) = ARBITRARY : bits(32);
        EDSCR().ITE = '1';                                // See comments in Write_EDITR()
    end;

    return value;
end;
```

G1.2.1.37 Read_DBGDTR_EL0

```
// Read_DBGDTR_EL0()
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

func Read_DBGDTR_EL0{N}() => bits(N)
begin
    // For MRS <Rt>, DBGDTRTX_EL0 N=32, X[t]=Zeros(32):result
    // For MRS <Xt>, DBGDTR_EL0 N=64, X[t]=result
    assert N IN {32, 64};
    var result : bits(N);
    if EDSCR().RXfull == '0' then
        result = ARBITRARY : bits(N);
    else
        // On a 64-bit read, implement a half-duplex channel
```

```

        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result[63:32] = DTRTX; end;
        result[31:0] = DTRRX;
    end;
    EDSCR().RXfull = '0';
    return result;
end;

```

G1.2.1.38 Write_DBGDTRRX_EL0

```

// Write_DBGDTRRX_EL0()
// =====
// Called on writes to debug register 0x080.

func Write_DBGDTRRX_EL0(memory_mapped : boolean, value : bits(32))
begin
    if EDSCR().ERR == '1' then return; end;           // Error flag set: ignore write

    if EDSCR().RXfull == '1' || (Halted() && EDSCR().MA == '1' && EDSCR().ITE == '0') then
        EDSCR().RXO = '1'; EDSCR().ERR = '1';        // Overrun condition: ignore write
        return;
    end;

    EDSCR().RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR().MA == '1' then
        EDSCR().ITE = '0';                          // See comments in Write_EDITR()
        ExecuteT32(0xEE10[15:0] /*hw1*/, 0x1E15[15:0] /*hw2*/); // T32 "MRS R1,DBGDTRRXint"
        ExecuteT32(0xF840[15:0] /*hw1*/, 0x1B04[15:0] /*hw2*/); // T32 "STR R1,[R0],#4"
        R(1) = ARBITRARY : bits(32);

        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR().ERR == '1' then
            EDSCR().RXfull = ARBITRARY : bit;
            DBGDTRRXint() = ARBITRARY : bits(32);
        else
            // "MRS X1,DBGDTRRX_EL0" calls Read_DBGDTR_EL0() which clears RXfull.
            assert EDSCR().RXfull == '0';
        end;

        EDSCR().ITE = '1';                          // See comments in Write_EDITR()
    end;
    return;
end;

```

G1.2.1.39 Write_DBGDTRTX_EL0

```

// Write_DBGDTRTX_EL0()
// =====
// Called on writes to debug register 0x08C.

func Write_DBGDTRTX_EL0(memory_mapped : boolean, value : bits(32))
begin
    DTRTX = value;
    return;
end;

```

G1.2.1.40 Write_DBGDTR_EL0

```

// Write_DBGDTR_EL0()
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

```

```
func Write_DBGDTR_EL0{N}(value_in : bits(N))
begin
    var value : bits(N) = value_in;
    // For MSR DBGDTRTX_EL0,<Rt> N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt> N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR().TXfull == '1' then
        value = ARBITRARY : bits(N);
    end;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value[63:32]; end;
    DTRTX = value[31:0]; // 32-bit or 64-bit write
    EDSCR().TXfull = '1';
    return;
end;
```

G1.2.1.41 Write_EDITR

```
// Write_EDITR()
// =====
// Called on writes to debug register 0x084.

func Write_EDITR(memory_mapped : boolean, value : bits(32))
begin
    if EDSCR().ERR == '1' then return; end; // Error flag set: ignore write

    if !Halted() then return; end; // Non-debug state: ignore write

    if EDSCR().ITE == '0' || EDSCR().MA == '1' then
        EDSCR().ITO = '1'; EDSCR().ERR = '1'; // Overrun condition: block write
        return;
    end;

    // ITE indicates whether the PE is ready to accept another instruction; the PE
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR().ITE = '0';

    ExecuteT32(value[15:0]/*hw1*/, value[31:16]/*hw2*/);

    EDSCR().ITE = '1';

    return;
end;
```

G1.2.1.42 DCPSInstruction

```
// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state

func DCPSInstruction(target_el : bits(2))
begin

    SynchronizeContext();

    var handle_el : bits(2);
    case target_el of
        when EL1 =>
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
                handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR().TGE == '1' then
                Undefined();
            end;
    end;
```

```

        else
            handle_el = EL1;
        end;
    when EL2 =>
        if !HaveEL(EL2) || EDSCR().HDD == '1' then
            Undefined();
        elsif PSTATE.EL == EL3 && !UsingAArch32() then
            handle_el = EL3;
        elsif CurrentSecurityState() == SS_Secure then
            Undefined();
        else
            handle_el = EL2;
        end;
    otherwise =>
        unreachable;
end;

if ELUsingAArch32(handle_el) then
    assert UsingAArch32(); // Cannot move from AArch64 to AArch32
    case handle_el of
        when EL1 =>
            AArch32_WriteMode(M32_Svc);
        when EL2 => AArch32_WriteMode(M32_Hyp);
        when EL3 =>
            AArch32_WriteMode(M32_Monitor);
    end;
    if handle_el == EL2 then
        ELR_hyp() = ARBITRARY : bits(32); HSR() = ARBITRARY : bits(32);
    else
        LR() = ARBITRARY : bits(32);
    end;
    SPSR_curr() = ARBITRARY : bits(32);
    if handle_el == EL2 then
        PSTATE.E = HSCTLR().EE;
    else
        PSTATE.E = SCTLR().EE;
    end;
    DLR() = ARBITRARY : bits(32); DSPSR() = ARBITRARY : bits(32);

else // Targeting AArch64
    PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
end;
UpdateEDSCRFields(); // Update EDSCR PE state flags
return;
end;

```

G1.2.1.43 DRPSInstruction

```

// DRPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

func DRPSInstruction()
begin
    SynchronizeContext();

    DebugRestorePSR();

    return;
end;

```

G1.2.1.44 DebugHalt

```
// DebugHalt
// =====
// Reason codes for entry to Debug state

constant DebugHalt_Breakpoint : bits(6)      = '000111';
constant DebugHalt_EDBGRQ : bits(6)         = '010011';
constant DebugHalt_Step_Normal : bits(6)     = '011011';
constant DebugHalt_Step_Exclusive : bits(6)  = '011111';
constant DebugHalt_OSUnlockCatch : bits(6)   = '100011';
constant DebugHalt_ResetCatch : bits(6)      = '100111';
constant DebugHalt_Watchpoint : bits(6)     = '101011';
constant DebugHalt_HaltInstruction : bits(6) = '101111';
constant DebugHalt_SoftwareAccess : bits(6)  = '110011';
constant DebugHalt_ExceptionCatch : bits(6)  = '110111';
constant DebugHalt_Step_NoSyndrome : bits(6) = '111011';
```

G1.2.1.45 DebugRestorePSR

```
// DebugRestorePSR()
// =====

func DebugRestorePSR()
begin
    // PSTATE.[N,Z,C,V,Q,GE,SS,D,A,I,F] are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        let spsr : bits(32) = SPSR_curr();
        SetPSTATEFromPSR{32}(spsr);
        PSTATE.[N,Z,C,V,Q,GE,SS,A,I,F] = ARBITRARY : bits(13);
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
        DLR() = ARBITRARY : bits(32); DSPSR() = ARBITRARY : bits(32);
    end;
    UpdateEDSCRFields(); // Update EDSCR PE state flags
end;
```

G1.2.1.46 DisableITRAndResumeInstructionPrefetch

```
// DisableITRAndResumeInstructionPrefetch()
// =====

impdef func DisableITRAndResumeInstructionPrefetch()
begin
    return;
end;
```

G1.2.1.47 ExecuteA64

```
// ExecuteA64()
// =====
// Execute an A64 instruction in Debug state.

impdef func ExecuteA64(instr : bits(32))
begin
    Unimplemented();
end;
```

G1.2.1.48 ExecuteT32

```
// ExecuteT32()
// =====
// Execute a T32 instruction in Debug state.

impdef func ExecuteT32(hw1 : bits(16), hw2 : bits(16))
begin
    Unimplemented();
end;
```

G1.2.1.49 ExitDebugState

```
// ExitDebugState()
// =====

func ExitDebugState()
begin
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR().STATUS = '000001'; // Signal restarting
    EDESR()[2:0] = '000'; // Clear any pending Halting debug events

    var new_pc : bits(64);
    var spsr : bits(64);

    if UsingAArch32() then
        new_pc = ZeroExtend{64}(DLR());
        spsr = ZeroExtend{64}(DSPSR());
    end;

    let illegal_psr_state : boolean = IllegalExceptionReturn{64}(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR{64}(spsr); // Can update privileged bits, even at EL0

    let branch_conditional : boolean = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc[0] = '0'; end;
        // AArch32 branch
        BranchTo{32}(new_pc[31:0], BranchType_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
        if illegal_psr_state && spsr[4] == '1' then
            new_pc[63:32] = ARBITRARY : bits(32);
            new_pc[1:0] = ARBITRARY : bits(2);
        end;
        // A type of branch that is never predicted
        BranchTo{64}(new_pc, BranchType_DBGEXIT, branch_conditional);
    end;

    // Atomically signal restarted
    EDSCR().STATUS = '000010';
    EDPRSR().SDR = '1';
    // End of atomically signal
    EDPRSR().HALTED = '0';
    UpdateEDSCRFields(); // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
end;
```

G1.2.1.50 Halt

```

// Halt()
// =====

func Halt(reason : bits(6))
begin
    let is_async : boolean = FALSE;
    Halt(reason, is_async);
end;

// Halt()
// =====

func Halt(reason : bits(6), is_async : boolean)
begin
    CTI_SignalEvent(CrossTriggerIn_CrossHalt); // Trigger other cores to halt

    let preferred_restart_address : bits(64) = ThisInstrAddr{}();
    var spsr : bits(64);
    if UsingAArch32() then
        spsr = ZeroExtend{64}{GetPSRFromPSTATE{32}(DebugState)};
    else
        spsr = GetPSRFromPSTATE{64}(DebugState);
    end;

    if UsingAArch32() then
        DLR() = preferred_restart_address[31:0];
        DSPSR() = spsr[31:0];
    else
        pass;
    end;
    EDSCR().ITE = '1';
    EDSCR().ITO = '0';
    if HaveEL(EL2) && (PSTATE.EL == EL2 || HCR().TGE == '1') then
        EDSCR().HDD = '0'; // If entered in Hyp or with TGE set, allow debug
    elseif HaveEL(EL2) then
        EDSCR().HDD = if ExternalHypInvasiveDebugEnabled() then '0' else '1';
    else
        EDSCR().HDD = '0';
    end;
    EDSCR().MA = '0';

    // In Debug state:
    // * PSTATE.[SS,SSBS,D,A,I,F] are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.[N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT] are also not observable, but since these
    //   are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.[IT,T] are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.[UAO,PAN] are observable and not changed on entry into Debug state.

    if UsingAArch32() then
        PSTATE.[IT,SS,A,I,F,T] = ARBITRARY : bits(13);
    else
        PSTATE.[SS,D,A,I,F] = ARBITRARY : bits(5);
    end;

    PSTATE.IL = '0';
    StopInstructionPrefetchAndEnableITR();
    // atomic write
    EDSCR().STATUS = reason;
    EDPSR().HALTED = '1';
    // end of atomic write
    UpdateEDSCRFields(); // Update EDSCR PE state flags.
    if !is_async then EndOfInstruction(EndOfInstructionReason_Exception); end;
    return;
end;

```

G1.2.1.51 HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

func HaltOnBreakpointOrWatchpoint() => boolean
begin
    return HaltingAllowed() && EDSCR().HDE == '1' && DBGOSLSR().OSLK == '0';
end;
```

G1.2.1.52 Halted

```
// Halted()
// =====

readonly func Halted() => boolean
begin
    return ! EDSCR().STATUS IN {'000001', '000010'};           // Halted
end;
```

G1.2.1.53 HaltingAllowed

```
// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

func HaltingAllowed() => boolean
begin
    if Halted() || DoubleLockStatus() then
        return FALSE;
    end;
    if HaveEL(EL2) && (PSTATE.EL == EL2 || HCR().TGE == '1') then
        return ExternalHypInvasiveDebugEnabled();
    end;
    return ExternalInvasiveDebugEnabled();
end;
```

G1.2.1.54 Restarting

```
// Restarting()
// =====

readonly func Restarting() => boolean
begin
    return EDSCR().STATUS == '000001';           // Restarting
end;
```

G1.2.1.55 StopInstructionPrefetchAndEnableITR

```
// StopInstructionPrefetchAndEnableITR()
// =====

impdef func StopInstructionPrefetchAndEnableITR()
begin
    return;
end;
```

G1.2.1.56 UpdateDbgAuthStatus

```
// UpdateDbgAuthStatus()
// =====
// Provides information about the state of the
// IMPLEMENTATION DEFINED authentication interface for debug.

func UpdateDbgAuthStatus()
begin
    var nsid, nsnid : bits(2);
    var hid, hnid : bits(2);
    if SecureOnlyImplementation() then
        nsid = '00';
    elseif ExternalInvasiveDebugEnabled() then
        nsid = '11';           // Non-secure Invasive debug implemented and enabled.
    else
        nsid = '10';           // Non-secure Invasive debug implemented and disabled.
    end;

    if SecureOnlyImplementation() then
        nsnid = '00';
    elseif ExternalNoninvasiveDebugEnabled() then
        nsnid = '11';          // Non-secure Non-Invasive debug implemented and enabled.
    else
        nsnid = '10';          // Non-secure Non-Invasive debug implemented and disabled.
    end;

    if ExternalHypNoninvasiveDebugEnabled() then
        hnid = '11';           // External Hyp non-invasive debug implemented and enabled.
    else
        hnid = '10';           // External Hyp non-invasive debug implemented and disabled.
    end;

    if ExternalHypInvasiveDebugEnabled() then
        hid = '11';            // External Hyp invasive debug implemented and enabled.
    else
        hid = '10';            // External Hyp invasive debug implemented and disabled.
    end;

    DBGAUTHSTATUS_EL1().NSID = nsid;
    DBGAUTHSTATUS_EL1().NSNID = nsnid;
    DBGAUTHSTATUS_EL1().SID = '00';
    DBGAUTHSTATUS_EL1().SNID = '00';
    DBGAUTHSTATUS_EL1().HNID = hnid;
    DBGAUTHSTATUS_EL1().HID = hid;
    return;
end;
```

G1.2.1.57 UpdateEDSCRFields

```
// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

func UpdateEDSCRFields()
begin
    if !Halted() then
        EDSCR().EL = '00';
    if HaveEL(EL2) then
        // HDD bit
        EDSCR().HDD = if ExternalHypInvasiveDebugEnabled() then '0' else '1';
    else
        // HDD bit
        EDSCR().HDD = '0';
    end;
end;
```

```

else
    EDSCR().EL = PSTATE.EL;
    // SError Pending.
    if EL2Enabled() && HCR().[AMO,TGE] == '10' && PSTATE.EL IN {EL0,EL1} then
        EDSCR().A = if IsVirtualErrorPending() then '1' else '0';
    else
        EDSCR().A = if IsPhysicalErrorPending() then '1' else '0';
    end;

    let ss : SecurityState = CurrentSecurityState();

end;
return;
end;

```

G1.2.1.58 haltingevents

```
var EDBGQR : Signal; // Input Signal in external debug interface
```

G1.2.1.59 CheckEDBGRQ

```

// CheckEDBGRQ()
// =====
// Checks IMPLEMENTATION DEFINED EDBGQR Input Signal of external debug interface.
// This is an example of an IMPLEMENTATION DEFINED source of External Debug Request debug events.

func CheckEDBGRQ()
begin
    if EDBGQR == HIGH then
        ExternalDebugRequest();
    end;
end;

```

G1.2.1.60 CheckExceptionCatch

```

// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level

func CheckExceptionCatch()
begin
    // Called after taking an exception, that is, such that the Security state and PSTATE.EL
    // are correct for the exception target. This function might also be called at any time.
    let base : integer{} = if SecurityStateAtEL(PSTATE.EL) == SS_Secure then 0 else 4;
    if HaltingAllowed() && EDECCR()[UInt(PSTATE.EL) + base] == '1' then
        Halt(DebugHalt_ExceptionCatch);
    end;
end;

```

G1.2.1.61 CheckExternalDebugRequestEvents

```

// CheckExternalDebugRequestEvents()
// =====
// Checks for all External Debug Request debug events.

func CheckExternalDebugRequestEvents()
begin
    CheckEDBGRQ();
    return;
end;

```

G1.2.1.62 CheckHaltingStep

```
// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

func CheckHaltingStep(is_async : boolean)
begin
    let step_enabled : boolean = EDECR().SS == '1' && HaltingAllowed();
    let active_pending : boolean = step_enabled && EDESR().SS == '1';
    if active_pending then
        Halt(DebugHalt_Step_NoSyndrome, is_async);
    end;
    if step_enabled then ShouldAdvanceHS = TRUE; end;
    return;
end;
```

G1.2.1.63 CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

func CheckOSUnlockCatch()
begin
    if EDECR().OSUCE == '1' then
        if !Halted() then EDESR().OSUC = '1'; end;
    end;
end;
```

G1.2.1.64 CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// =====
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

func CheckPendingOSUnlockCatch()
begin
    if HaltingAllowed() && EDESR().OSUC == '1' then
        let is_async : boolean = TRUE;
        Halt(DebugHalt_OSUnlockCatch, is_async);
    end;
end;
```

G1.2.1.65 CheckPendingResetCatch

```
// CheckPendingResetCatch()
// =====
// Check whether EDESR.RC has been set by a Reset Catch debug event

func CheckPendingResetCatch()
begin
    if HaltingAllowed() && EDESR().RC == '1' then
        let is_async : boolean = TRUE;
        Halt(DebugHalt_ResetCatch, is_async);
    end;
end;
```

G1.2.1.66 CheckResetCatch

```
// CheckResetCatch()
// =====
// Called after reset

func CheckResetCatch()
begin
    if EDECR().RCE == '1' then
        EDESR().RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch); end;
    end;
end;
```

G1.2.1.67 CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====
// Check for access to Breakpoint and Watchpoint registers.

func CheckSoftwareAccessToDebugRegisters()
begin
    let os_lock : bit = DBGOSLSR().OSLK;
    if HaltingAllowed() && EDSCR().TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
    end;
end;
```

G1.2.1.68 ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

func ExternalDebugRequest()
begin
    if HaltingAllowed() then
        let is_async : boolean = TRUE;
        Halt(DebugHalt_EDBGRQ, is_async);
        // Otherwise the CTI continues to assert the debug request until it is taken.
    end;
end;
```

G1.2.1.69 HSAdvance

```
// HSAdvance()
// =====
// Advance the Halting Step State Machine

func HSAdvance()
begin
    if !ShouldAdvanceHS then return; end;
    let step_enabled : boolean = EDECR().SS == '1' && HaltingAllowed();
    let active_not_pending : boolean = step_enabled && EDESR().SS == '0';
    if active_not_pending then EDESR().SS = '1'; end; // set as pending.
    ShouldAdvanceHS = FALSE;
    return;
end;
```

G1.2.1.70 ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'.

func ExternalDebugInterruptsDisabled(target : bits(2)) => boolean
begin
    var int_dis : boolean;
    let ss : SecurityState = SecurityStateAtEL(target);
    case target of
        when EL3 =>
            int_dis = (EDSCR().INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
        when EL2 =>
            int_dis = (EDSCR().INTdis == '1x' && ExternalInvasiveDebugEnabled() &&
                ExternalHypInvasiveDebugEnabled());
        when EL1 =>
            if ss == SS_Secure then
                int_dis = (EDSCR().INTdis == '1x' && ExternalSecureInvasiveDebugEnabled());
            else
                int_dis = (EDSCR().INTdis != '00' && ExternalInvasiveDebugEnabled());
            end;
    end;
    return int_dis;
end;
```

G1.2.1.71 pmu

```
var PMUEventAccumulator : array [[31]] of integer; // Accumulates PMU events for a cycle

// Constant used in PMU functions to represent actions on the cycle counter.
constant CYCLE_COUNTER_ID : integer = 31;
```

G1.2.1.72 ApplyPMUZeroMask

```
// ApplyPMUZeroMask()
// =====
// Zero the set of counters specified by the mask in 'masked_val'.
// 'masked_val' is expected to already be masked for the permitted counters for the current
// access type before this function is called.

func ApplyPMUZeroMask(masked_val : bits(64))
begin
    for idx = 0 to 63 do
        if masked_val[idx] == '1' then
            case idx of
                when CYCLE_COUNTER_ID =>
                    PMCCNTR() = Zeros{64};
                otherwise =>
                    PMEVCNTR(idx) = Zeros{32};
            end;
        end;
    end;
    return;
end;
```

G1.2.1.73 CheckForPMUOverflow

```
// CheckForPMUOverflow()
// =====
// Called before each instruction is executed.
// If a PMU event counter has overflowed, this function might do any of:
// - Signal a Performance Monitors overflow interrupt request.
// - Signal a CTI Performance Monitors overflow event.
```

```
func CheckForPMUOverflow()
begin
    let include_r1 : boolean = TRUE;
    let include_r2 : boolean = TRUE;

    let pmuirq : boolean = CheckPMUOverflowCondition(PMUOverflowCondition_IRQ,
                                                    include_r1, include_r2);

    SetInterruptRequestLevel(InterruptID_PMUIRQ,
                            if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow,
                    if pmuirq then HIGH else LOW);

    return;
end;
```

G1.2.1.74 CheckPMUOverflowCondition

```
// CheckPMUOverflowCondition()
// =====
// Checks for PMU overflow under certain parameter conditions described by 'reason'.
// If 'include_r1' is TRUE, then check counters in the range [0..(HPMN-1)] and CCNTR
// unless excluded by 'reason'.
// If 'include_r2' is TRUE, then check counters in the range [HPMN..(N-1)].

func CheckPMUOverflowCondition(reason : PMUOverflowCondition,
                              include_r1 : boolean, include_r2 : boolean) => boolean
begin
    // 'reason' is decoded into a further set of parameters:
    // If 'check_e' is TRUE, then check the applicable one of PMCR_EL0.E and MDCR_EL2.HPME.
    // If 'check_inten' is TRUE, then check the applicable PMINTENCLR_EL1 bit.
    // If 'exclude_cyc' is TRUE, then CCNTR is NOT checked.
    var check_e : boolean;
    var check_inten : boolean;
    var exclude_cyc : boolean;
    case reason of
        when PMUOverflowCondition_IRQ =>
            check_e = TRUE;
            check_inten = TRUE;
            exclude_cyc = FALSE;
        otherwise =>
            unreachable;
    end;

    var ovsf : bits(64);

    ovsf = ZeroExtend{64}(PMOVSSET());

    let counters : integer{} = NUM_PMU_COUNTERS;
    // Remove unimplemented counters - these fields are RES0
    if counters < 31 then
        ovsf[30:counters] = Zeros{31-counters};
    end;

    for idx = 0 to counters - 1 do
        var global_en : bit;
        case GetPMUCounterRange(idx) of
            when PMUCounterRange_R1 =>
                global_en = if HaveAArch64() then PMCR_EL0().E else PMCR().E;
                if !include_r1 then
                    ovsf[idx] = '0';
                end;
            when PMUCounterRange_R2 =>
                global_en = HDCR().HPME;
                if !include_r2 then
```

```

        ovsf[idx] = '0';
    end;
    otherwise =>
        unreachable;
    end;
    if check_e then
        ovsf[idx] = ovsf[idx] AND global_en;
    end;
end;

// Cycle counter
if exclude_cyc || !include_r1 then
    ovsf[CYCLE_COUNTER_ID] = '0';
end;

if check_e then
    ovsf[CYCLE_COUNTER_ID] = ovsf[CYCLE_COUNTER_ID] AND PMCR_EL0().E;
end;

if check_inten then
    let inten : bits(64) = ZeroExtend{}(PMINTENCLR());
    ovsf = ovsf AND inten;
end;

return !IsZero(ovsf);
end;

```

G1.2.1.75 CountPMUEvents

```

// CountPMUEvents()
// =====
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID (31).

func CountPMUEvents(idx : integer) => boolean
begin
    let counters : integer = NUM_PMU_COUNTERS;
    assert idx == CYCLE_COUNTER_ID || idx < counters;

    var debug : boolean;
    var enabled : boolean;
    var prohibited : boolean;
    var filtered : boolean;

    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    let counter_range : PMUCounterRange = GetPMUCounterRange(idx);
    var ss : SecurityState = CurrentSecurityState();

    // Main enable controls
    var global_en : bit;
    var counter_en : bit;
    case counter_range of
        when PMUCounterRange_R1 =>
            global_en = if HaveAArch64() then PMCR_EL0().E else PMCR().E;
        when PMUCounterRange_R2 =>
            global_en = HDCR().HPME;
        otherwise =>
            unreachable;
    end;

    case idx of
        when CYCLE_COUNTER_ID =>
            counter_en = if HaveAArch64() then PMCNTENSET_EL0().C else PMCNTENSET().C;

```



```

        otherwise =>
            counter_en = if HaveAArch64() then PMCNTENSET_EL0()[idx] else PMCNTENSET()[idx];
        end;

    enabled = global_en == '1' && counter_en == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is permitted as EL3 is not implemented
    // Event counting at EL2 is prohibited if all of:
    // * PMNx is not reserved for EL2
    // * HDCR.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && counter_range == PMUCounterRange_R1 then
        prohibited = HDCR().HPMD == '1';
    end;
    // The IMPLEMENTATION DEFINED authentication interface might override software
    if ExternalSecureNoninvasiveDebugEnabled() then
        prohibited = FALSE;
    end;

    // PMCR_EL0.DP or PMCR.DP disables the cycle counter when event counting is prohibited
    if prohibited && idx == CYCLE_COUNTER_ID then
        let dp : bit = PMCR().DP;
        enabled = enabled && dp == '0';
        // Otherwise whether event counting is prohibited does not affect the cycle counter
        prohibited = FALSE;
    end;

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
    var filter : bits(32);
    case idx of
        when CYCLE_COUNTER_ID =>
            filter = if HaveAArch64() then PMCCFILTR_EL0()[31:0] else PMCCFILTR();
            otherwise =>
                filter = if HaveAArch64() then PMEVTYPER_EL0(idx)[31:0] else PMEVTYPER(idx);
    end;

    let p : bit = filter[31];
    let u : bit = filter[30];
    let nsh : bit = if HaveEL(EL2) then filter[27] else '0';

    case PSTATE.EL of
        when EL0 =>
            filtered = u == '1';
        when EL1 =>
            filtered = p == '1';
        when EL2 =>
            filtered = nsh == '0';
    end;

    return !debug && enabled && !prohibited && !filtered;
end;

```

G1.2.1.76 EffectiveHPMN

```

// EffectiveHPMN()
// =====
// Returns the Effective value of MDCR_EL2.HPMN or HDCR.HPMN.

func EffectiveHPMN() => bits(5)
begin
    let counters : integer = NUM_PMU_COUNTERS;
    var hpmn_bits : bits(5);

    if HaveEL(EL2) then // Software can reserve some event counters for EL2

```

```
hpmn_bits = HDCR().HPMN;

if UInt(hpmn_bits) > counters || IsZero(hpmn_bits) then
    (-, hpmn_bits) = ConstrainUnpredictableBits{5}(Unpredictable_RES_HPMN);
end;
else
    hpmn_bits = counters[4:0];
end;

return hpmn_bits;
end;
```

G1.2.1.77 ExtRegGetPMUAccessMask

```
// ExtRegGetPMUAccessMask()
// =====
// Return a mask of the PMU counters accessible via external PMU access.

func ExtRegGetPMUAccessMask(addrdesc : AddressDescriptor) => bits(64)
begin
    var mask : bits(64) = Zeros{};

    mask[CycleCounterID] = '1';

    let counters : integer{} = NUM_PMU_COUNTERS;

    if counters > 0 then
        mask[counters-1:0] = Ones{counters};
    end;

    return mask;
end;
```

G1.2.1.78 ExtRegZeroCycleCounter

```
// ExtRegZeroCycleCounter()
// =====
// Called on an external PMU write to PMCR_EL0 that writes '1' to PMCR_EL0.C.

func ExtRegZeroCycleCounter(addrdesc : AddressDescriptor)
begin
    var mask : bits(64) = Zeros{};
    mask[CycleCounterID] = '1';
    ExtRegZeroPMUCounters(mask, addrdesc);
end;
```

G1.2.1.79 ExtRegZeroEventCounters

```
// ExtRegZeroEventCounters()
// =====
// Zero all the event counters.
// Called on an external PMU write to PMCR_EL0 that writes '1' to PMCR_EL0.P.

func ExtRegZeroEventCounters(addrdesc : AddressDescriptor)
begin
    ExtRegZeroPMUCounters(Zeros{33} :: Ones{31}, addrdesc);
end;
```

G1.2.1.80 ExtRegZeroPMUCounters

```
// ExtRegZeroPMUCounters()
// =====
// Zero set of counters specified by the mask in 'val', for external PMU access.

func ExtRegZeroPMUCounters(val : bits(64), addrdesc : AddressDescriptor)
begin
    let masked_val : bits(64) = val AND ExtRegGetPMUAccessMask(addrdesc);
    ApplyPMUZeroMask(masked_val);
    return;
end;
```

G1.2.1.81 GetNumEventCountersAccessible

```
// GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

func GetNumEventCountersAccessible() => integer{0..NUM_PMU_COUNTERS}
begin
    var n : integer;

    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(EffectiveHPMN());
    else
        n = NUM_PMU_COUNTERS;
    end;

    return n as integer{0..NUM_PMU_COUNTERS};
end;
```

G1.2.1.82 GetNumEventCountersSelfHosted

```
// GetNumEventCountersSelfHosted()
// =====
// Return the number of event counters that can be accessed by the Self-hosted software.

func GetNumEventCountersSelfHosted() => integer
begin
    return NUM_PMU_COUNTERS;
end;
```

G1.2.1.83 GetPMUCounterRange

```
// GetPMUCounterRange()
// =====
// Returns the range that a counter is currently in.

func GetPMUCounterRange(n : integer) => PMUCounterRange
begin

    let counters : integer = NUM_PMU_COUNTERS;
    let hpmn : integer = UInt(EffectiveHPMN());

    if n < hpmn then
        return PMUCounterRange_R1;
    elseif n < counters then
        return PMUCounterRange_R2;
    elseif n == CYCLE_COUNTER_ID then
        return PMUCounterRange_R1;
    else
        unreachable;
    end;
end;
```

G1.2.1.84 GetPMUReadMask

```
// GetPMUReadMask()
// =====
// Return a mask of the PMU counters that can be read at the current
// Exception level.
// This mask masks reads from PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, and PMOVSCCLR_EL0.

func GetPMUReadMask() => bits(64)
begin
    var mask : bits(64) = SysRegGetPMUAccessMask();
    return mask;
end;
```

G1.2.1.85 GetPMUWriteMask

```
// GetPMUWriteMask()
// =====
// Return a mask of the PMU counters writable at the current Exception level.
// This mask masks writes to PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, and PMOVSCCLR_EL0.

func GetPMUWriteMask() => bits(64)
begin
    var mask : bits(64) = SysRegGetPMUAccessMask();

    return mask;
end;
```

G1.2.1.86 HasElapsed64Cycles

```
// HasElapsed64Cycles()
// =====
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.

impdef func HasElapsed64Cycles() => boolean
begin
    return TRUE;
end;
```

G1.2.1.87 PMUCounterRange

```
// PMUCounterRange
// =====
// Enumerates the ranges to which an event counter belongs to.

type PMUCounterRange of enumeration {
    PMUCounterRange_R1,
    PMUCounterRange_R2
};
```

G1.2.1.88 PMUEvent

```
// PMUEvent()
// =====
// Generate a PMU event. By default, increment by 1.
func PMUEvent(pmuevent : bits(16))
begin
    PMUEvent(pmuevent, 1);
end;

// PMUEvent()
// =====
```

```
// Accumulate a PMU Event.

func PMUEvent(pmuevent : bits(16), increment : integer)
begin
    let counters : integer = NUM_PMU_COUNTERS;
    if counters != 0 then
        for idx = 0 to counters - 1 do
            PMUEvent(pmuevent, increment, idx);
        end;
    end;

end;

// PMUEvent()
// =====
// Accumulate a PMU Event for a specific event counter.

func PMUEvent(pmuevent : bits(16), increment : integer, idx : integer)
begin
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    end;
    if ZeroExtend{16}(PMEVTYPER(idx).evtCount) == pmuevent then
        PMUEventAccumulator[[idx]] = PMUEventAccumulator[[idx]] + increment;
    end;
end;
```

G1.2.1.89 PMUOverflowCondition

```
// PMUOverflowCondition()
// =====
// Enumerates the reasons for which the PMU overflow condition is evaluated.

type PMUOverflowCondition of enumeration {
    PMUOverflowCondition_IRQ,
};
```

G1.2.1.90 PMUSwIncrement

```
// PMUSwIncrement()
// =====
// Generate PMU Events on a System register write to PMSWINC

func PMUSwIncrement(sw_incr_in : bits(64))
begin

    var sw_incr : bits(64) = sw_incr_in;
    sw_incr = sw_incr AND SysRegGetPMUAccessMask();
    for idx = 0 to 30 do
        if sw_incr[idx] == '1' then
            PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
        end;
    end;

    return;
end;
```

G1.2.1.91 SysRegGetPMUAccessMask

```
// SysRegGetPMUAccessMask()
// =====
// Return a mask of the PMU counters accessible at the current Exception level
// for a System register access
```

```
func SysRegGetPMUAccessMask() => bits(64)
begin
    var mask : bits(64) = Zeros{};

    // PMCNTR_ELO is always implemented and accessible
    mask[CYCLE_COUNTER_ID] = '1';

    // PMEVCNTR_ELO(n)
    let counters : integer{} = GetNumEventCountersAccessible();
    if counters > 0 then
        mask[counters-1:0] = Ones{counters};
    end;

    return mask;
end;
```

G1.2.1.92 SysRegZeroCycleCounter

```
// SysRegZeroCycleCounter()
// =====
// Called on a System register write to PMCR that writes '1' to PMCR.C.

func SysRegZeroCycleCounter()
begin
    var mask : bits(64) = Zeros{};
    mask[CYCLE_COUNTER_ID] = '1';
    SysRegZeroPMUCounters(mask);
end;
```

G1.2.1.93 SysRegZeroEventCounters

```
// SysRegZeroEventCounters()
// =====
// Zero all the event counters.
// Called on a System register write to PMCR that writes '1' to PMCR.P.

func SysRegZeroEventCounters()
begin
    SysRegZeroPMUCounters(Zeros{33} :: Ones{31});
end;
```

G1.2.1.94 SysRegZeroPMUCounters

```
// SysRegZeroPMUCounters()
// =====
// Zero set of counters specified by the mask in 'val', for System register access.

func SysRegZeroPMUCounters(val : bits(64))
begin
    let masked_val : bits(64) = val AND GetPMUWriteMask();
    ApplyPMUZeroMask(masked_val);
    return;
end;
```

G1.2.1.95 CreatePCSample

```
// CreatePCSample()
// =====

func CreatePCSample()
begin
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.
```

```

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr{64}();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = CONTEXTIDR();
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        pc_sample.vmid = VSCTLR().VMID;
    end;
    return;
end;

```

G1.2.1.96 PCSample

```

// PCSample
// =====

type PCSample of record {
    valid : boolean,
    pc : bits(64),
    el : bits(2),
    rw : bit,
    ss : SecurityState,
    has_el2 : boolean,
    contextidr : bits(32),
    vmid : bits(NUM_VMIDBITS)
};

var pc_sample : PCSample;

```

G1.2.1.97 Read_EDPCSRlo

```

// Read_EDPCSRlo()
// =====

func Read_EDPCSRlo(memory_mapped : boolean) => bits(32)
begin
    // The Software lock is OPTIONAL.
    let update : boolean = !memory_mapped || EDLSR().SLK == '0'; // Software locked: no side-effects
    var sample : bits(32);
    if pc_sample.valid then
        sample = pc_sample.pc[31:0];
        if update then
            EDPCSRhi = (if pc_sample.rw == '0' then Zeros{32} else pc_sample.pc[63:32]);
            EDCIDSR() = pc_sample.contextidr;
            EDVIDSR().VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1, EL0}
                               then pc_sample.vmid else Zeros{NUM_VMIDBITS});
            EDVIDSR().E2 = (if pc_sample.el == EL2 then '1' else '0');
            // The conditions for setting HV are not specified if PCSRhi is zero.
            // An example implementation may be "pc_sample.rw".
        end;
    else
        sample = Ones{32};
        if update then
            EDPCSRhi = ARBITRARY : bits(32);
            EDCIDSR() = ARBITRARY : bits(32);
            EDVIDSR() = (ARBITRARY : bits(4))::Zeros{20}::(ARBITRARY : bits(8));
        end;
    end;

    return sample;
end;

```

G1.2.1.98 DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

func DebugExceptionReturnSS{N}(spsr : bits(N)) => bit
begin
    assert Halted() || Restarting() || PSTATE.EL != EL0;

    var enabled_at_source : boolean;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32_GenerateDebugExceptions();
    end;
    var valid : boolean;
    var dest_el : bits(2);
    if IllegalExceptionReturn{N}(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR{N}(spsr); assert valid;
    end;

    let dest_ss : SecurityState = SecurityStateAtEL(dest_el);
    var enabled_at_dest : boolean;
    let dest_using_32 : boolean = (if dest_el == EL0 then spsr[4] == '1'
                                   else ELUsingAArch32(dest_el));
    if dest_using_32 then
        enabled_at_dest = AArch32_GenerateDebugExceptionsFrom(dest_el, dest_ss);
    end;

    let ELd : bits(2) = DebugTargetFrom(dest_ss);
    var SS_bit : bit;
    SS_bit = '0';
    return SS_bit;
end;
```

G1.2.1.99 DataCacheWatchpointSize

```
// DataCacheWatchpointSize()
// =====
// Return the IMPLEMENTATION DEFINED data cache watchpoint size
func DataCacheWatchpointSize() => integer
begin
    let size : integer = ImpDefInt("Data Cache Invalidate Watchpoint Size");
    assert IsPow2(size) && size >= 2^(UInt(CTR().DminLine) + 2) && size <= 2048;
    return size;
end;
```

G1.2.1.100 WatchpointInfo

```
// WatchpointInfo
// =====
// Watchpoint related fields.

type WatchpointInfo of record {
    value_match : boolean,          // Watchpoint match
    vaddress : bits(64)             // Matching Virtual Address
};
```

G1.2.2 shared/exceptions

This section includes the following pseudocode functions:

- [EffectiveHCR_AMO](#)
- [PhysicalSErrorTarget](#)
- [SyncExternalAbortTarget](#)
- [ConditionSyndrome](#)
- [Exception](#)
- [ExceptionRecord](#)
- [ExceptionSyndrome](#)
- [Undefined](#)

G1.2.2.1 EffectiveHCR_AMO

```
// EffectiveHCR_AMO()
// =====
// Return the Effective value of HCR_EL2.AMO.

func EffectiveHCR_AMO() => bit
begin
    if EffectiveTGE() == '1' then
        return '1';
    elsif EL2Enabled() then
        return HCR().AMO;
    else
        return '0';
    end;
end;
```

G1.2.2.2 PhysicalSErrorTarget

```
// PhysicalSErrorTarget()
// =====
// Returns a tuple of whether SError exception can be taken and, if so, the
// target Exception level.

func PhysicalSErrorTarget() => (boolean, bits(2))
begin
    if Halted() then
        return (TRUE, ARBITRARY : bits(2));
    end;

    let effective_ea : bit = '0';           // EL3 is not implemented
    let effective_amo : bit = EffectiveHCR_AMO();
    let effective_tge : bit = EffectiveTGE();

    // The exception is masked by software.
    var masked : boolean;
    case PSTATE.EL of
        when EL2 =>
            masked = (effective_ea == '0' &&
                ((!UsingAArch32() && effective_tge == '0' && effective_amo == '0') ||
                 PSTATE.A == '1'));
        when EL1, EL0 =>
            masked = (effective_ea == '0' && effective_amo == '0' && PSTATE.A == '1');
    end;

    // External debug might disable the exception in the current Security state.
    // This is not relevant at EL3.
    let intdis : boolean = PSTATE.EL != EL3 && ExternalDebugInterruptsDisabled(EL1);
```

```

var target_el : bits(2) = ARBITRARY : bits(2);

if EL2Enabled() && effective_amo == '1' && !intdis && PSTATE.EL IN {EL0, EL1} then
    target_el = EL2;
    masked = FALSE;

elseif PSTATE.EL == EL2 then
    if !masked then target_el = EL2; end;

else
    assert (PSTATE.EL == EL1 ||
           PSTATE.EL == EL0);
    if !masked then target_el = EL1; end;
end;

// External debug might disable the exception for the target Exception level.
if !masked && ExternalDebugInterruptsDisabled(target_el) then
    masked = TRUE;
    target_el = ARBITRARY : bits(2);
end;

return (masked, target_el);
end;

```

G1.2.2.3 SyncExternalAbortTarget

```

// SyncExternalAbortTarget()
// =====
// Returns the target Exception level for a Synchronous External Data or
// Instruction or Prefetch Abort.

func SyncExternalAbortTarget(fault : FaultRecord) => bits(2)
begin
    let effective_tge : bit = EffectiveTGE();

    var target_el : bits(2);

    if (EL2Enabled() && PSTATE.EL IN {EL1, EL0} &&
        (IsSecondStage(fault) ||
         (PSTATE.EL == EL0 && effective_tge == '1'))) then
        target_el = EL2;

    else
        assert PSTATE.EL != EL3 || ELUsingAArch32(EL3);
        target_el = (if PSTATE.EL == EL2 then EL2 else EL1);
    end;

    return target_el;
end;

```

G1.2.2.4 ConditionSyndrome

```

// ConditionSyndrome()
// =====
// Return CV and COND fields of instruction syndrome

func ConditionSyndrome() => bits(5)
begin

    var syndrome : bits(5);

```

```

if UsingAArch32() then
    let cond = CurrentCond();
    if PSTATE.T == '0' then // A32
        syndrome[4] = '1';
        // A conditional A32 instruction that is known to pass its condition code check
        // can be presented either with COND set to 0xE, the value for unconditional, or
        // the COND value held in the instruction.
        if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
            syndrome[3:0] = '1110';
        else
            syndrome[3:0] = cond;
        end;
    else // T32
        // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
        // * CV set to 0 and COND is set to an UNKNOWN value
        // * CV set to 1 and COND is set to the condition code for the condition that
        // applied to the instruction.
        if ImpDefBool("Condition valid for trapped T32") then
            syndrome[4] = '1';
            syndrome[3:0] = cond;
        else
            syndrome[4] = '0';
            syndrome[3:0] = ARBITRARY : bits(4);
        end;
    end;
else
    syndrome[4] = '1';
    syndrome[3:0] = '1110';
end;
return syndrome;
end;

```

G1.2.2.5 Exception

```

// Exception
// =====
// Classes of exception.

type Exception of enumeration {
    Exception_Uncategorized, // Uncategorized or unknown reason
    Exception_WFxTrap, // Trapped WFI or WFE instruction
    Exception_CP15RTTTrap, // Trapped AArch32 MCR or MRC access, coproc=0b111
    Exception_CP15RRTTrap, // Trapped AArch32 MCRR or MRRC access, coproc=0b1111
    Exception_CP14RTTTrap, // Trapped AArch32 MCR or MRC access, coproc=0b1110
    Exception_CP14DTTTrap, // Trapped AArch32 LDC or STC access, coproc=0b1110
    Exception_CP14RRTTrap, // Trapped AArch32 MRRC access, coproc=0b1110
    Exception_AdvSIMDFAccessTrap, // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap, // Trapped access to SIMD or FP ID register
    Exception_OtherTrap, // Trapped access to other instructions
    // Trapped BXJ instruction not supported in Armv8
    Exception_IllegalState, // Illegal Execution state
    Exception_SupervisorCall, // Supervisor Call
    Exception_HypervisorCall, // Hypervisor Call
    Exception_MonitorCall, // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap, // Trapped MRS or MSR System register access
    Exception_InstructionAbort, // Instruction Abort or Prefetch Abort
    Exception_PCAalignment, // PC alignment fault
    Exception_DataAbort, // Data Abort
    Exception_SPAalignment, // SP alignment fault
    Exception_FPtrappedException, // IEEE trapped FP exception
    Exception_SError, // SError interrupt
    Exception_Breakpoint, // (Hardware) Breakpoint
    Exception_SoftwareStep, // Software Step
    Exception_Watchpoint, // Watchpoint
    Exception_SoftwareBreakpoint, // Software Breakpoint Instruction
    Exception_VectorCatch, // AArch32 Vector Catch
}

```

```
Exception_IRQ,           // IRQ interrupt
Exception_FIQ};          // FIQ interrupt
```

G1.2.2.6 ExceptionRecord

```
// ExceptionRecord
// =====

type ExceptionRecord of record {
    exceptype : Exception,           // Exception class
    syndrome : IssType,               // Syndrome record
    vaddress : bits(64),              // Virtual fault address
    ipavalid : boolean,              // Validity of Intermediate Physical fault address
    ipaddress : bits(NUM\_PABITS),      // Intermediate Physical fault address
    trappedsyscallinst : boolean;    // Trapped SVC or SMC instruction
```

G1.2.2.7 ExceptionSyndrome

```
// ExceptionSyndrome()
// =====
// Return a blank exception syndrome record for an exception of the given type.

func ExceptionSyndrome(exceptype : Exception) => ExceptionRecord
begin

    var r : ExceptionRecord;

    r.exceptype      = exceptype;

    // Initialize all other fields
    r.syndrome.iss   = Zeros{25};
    r.syndrome.iss2  = Zeros{24};
    r.vaddress       = Zeros{64};
    r.ipavalid       = FALSE;
    r.ipaddress      = Zeros{NUM\_PABITS};
    r.trappedsyscallinst = FALSE;
    return r;
end;
```

G1.2.2.8 Undefined

```
// Undefined()
// =====

noreturn func Undefined()
begin
    AArch32\_Undefined();
end;
```

G1.2.3 shared/functions

This section includes the following pseudocode functions:

- [EncodeLDFSC](#)
- [IPAValid](#)
- [IsAsyncAbort](#)
- [IsDebugException](#)
- [IsExternalAbort](#)
- [IsExternalAbortOnWalk](#)
- [IsExternalSyncAbort](#)

- [IsFault](#)
- [IsErrorInterrupt](#)
- [IsSecondStage](#)
- [LSInstructionSyndrome](#)
- [CACHE_OP](#)
- [CPASAtPAS](#)
- [CPASAtSecurityState](#)
- [CacheRecord](#)
- [DecodeSW](#)
- [GetCacheInfo](#)
- [IsZeroBit](#)
- [NormalizeReal](#)
- [RShr](#)
- [Reverse](#)
- [Signal](#)
- [counters](#)
- [AArch32_CheckTimerConditions](#)
- [CNTHCTL_EL2_VHE](#)
- [GenericCounterTick](#)
- [IsTimerConditionMet](#)
- [SetEventRegister](#)
- [TestEventCNTP](#)
- [TestEventCNTV](#)
- [BitReverse](#)
- [Poly32Mod2](#)
- [AESInvMixColumns](#)
- [AESInvShiftRows](#)
- [AESInvSubBytes](#)
- [AESMixColumns](#)
- [AESShiftRows](#)
- [AESSubBytes](#)
- [FFmul02](#)
- [FFmul03](#)
- [FFmul09](#)
- [FFmul0B](#)
- [FFmul0D](#)
- [FFmul0E](#)
- [SHA256hash](#)
- [SHA256hashSIGMA0](#)
- [SHA256hashSIGMA1](#)
- [SHAchoose](#)
- [SHAmajority](#)
- [SHAparity](#)
- [DecodeType](#)
- [EndOfDecode](#)
- [ClearExclusiveByAddress](#)
- [ClearExclusiveLocal](#)

- ExclusiveMonitorsStatus
- IsExclusiveGlobal
- IsExclusiveLocal
- MarkExclusiveGlobal
- MarkExclusiveLocal
- ProcessorID
- HaveSoftwareLock
- HaveTraceExt
- ClearPendingPhysicalSError
- ClearPendingVirtualSError
- ExternalAbort
- ExternalFault
- GetPendingPhysicalSError
- HandleExternalAbort
- HandleExternalReadAbort
- HandleExternalTTWAbort
- HandleExternalWriteAbort
- IsExternalAbortTakenSynchronously
- IsPhysicalSErrorPending
- IsSErrorEdgeTriggered
- IsSynchronizablePhysicalSErrorPending
- IsVirtualSErrorPending
- PendSErrorInterrupt
- FixedToFP
- FPAbs
- FPAdd
- FPBits
- FPBitsType
- FPFracBits
- FPCompare
- FPCompareEQ
- FPCompareGE
- FPCompareGT
- FPConvert
- FPConvertNaN
- FPDecodeRM
- FPDecodeRounding
- FPDefaultNaN
- FPDiv
- FPExc
- FPInfinity
- FPMax
- FPMaxNormal
- FPMaxNum
- IsMerging
- FPMin
- FPMinNum

- [FPMul](#)
- [FPMulAdd](#)
- [FPMulX](#)
- [FPNeg](#)
- [FPOnePointFive](#)
- [FPProcessException](#)
- [FPProcessNaN](#)
- [FPProcessNaNs](#)
- [FPProcessNaNs3](#)
- [FPRecipEstimate](#)
- [RecipEstimate](#)
- [FPRecpX](#)
- [FPRound](#)
- [FPRounding](#)
- [FPRoundingMode](#)
- [FPRoundInt](#)
- [FPRSqrtEstimate](#)
- [RecipSqrtEstimate](#)
- [FPSqrt](#)
- [FPSub](#)
- [FPThree](#)
- [FPToFixed](#)
- [FPTwo](#)
- [FPType](#)
- [FPUnpack](#)
- [FPZero](#)
- [VFPEExpandImm](#)
- [AddWithCarry](#)
- [InterruptID](#)
- [SetInterruptRequestLevel](#)
- [AccessDescriptor](#)
- [AccessType](#)
- [AddrTop](#)
- [AddressSize](#)
- [AlignmentEnforced](#)
- [AllInAlignedQuantity](#)
- [Allocation](#)
- [BigEndian](#)
- [BigEndianReverse](#)
- [CacheOp](#)
- [CacheOpScope](#)
- [CachePASpace](#)
- [CacheType](#)
- [Cacheability](#)
- [CreateAccDescA32LSMD](#)
- [CreateAccDescASIMD](#)
- [CreateAccDescAT](#)

- [CreateAccDescAcqRel](#)
- [CreateAccDescDC](#)
- [CreateAccDescDCZero](#)
- [CreateAccDescExLDST](#)
- [CreateAccDescGPR](#)
- [CreateAccDescIC](#)
- [CreateAccDescIFetch](#)
- [CreateAccDescS1TTW](#)
- [CreateAccDescS2TTW](#)
- [DataMemoryBarrier](#)
- [DataSynchronizationBarrier](#)
- [DeviceType](#)
- [Fault](#)
- [FaultRecord](#)
- [FullAddress](#)
- [Hint_Prefetch](#)
- [IsWatchpointableAccess](#)
- [MBMaintenanceScope](#)
- [MBReqTypes](#)
- [MPURecord](#)
- [MemAttrHints](#)
- [MemOp](#)
- [MemType](#)
- [MemoryAttributes](#)
- [NewAccDesc](#)
- [PASpace](#)
- [Permissions](#)
- [PhysMemRead](#)
- [PhysMemRetStatus](#)
- [PhysMemWrite](#)
- [PrefetchHint](#)
- [S1AccessControls](#)
- [S2AccessControls](#)
- [Shareability](#)
- [SpeculativeStoreBypassBarrierToPA](#)
- [SpeculativeStoreBypassBarrierToVA](#)
- [VARange](#)
- [ASID](#)
- [BranchTo](#)
- [BranchToAddr](#)
- [BranchType](#)
- [Branchtypetaken](#)
- [ESize](#)
- [EffectiveFPCR](#)
- [Hint_Branch](#)
- [NextInstrAddr](#)
- [ResetExternalDebugRegisters](#)

- ThisInstrAddr
- UnimplementedIDRegister
- V
- Vpart
- _PC
- _R
- _V
- SPSR_curr
- AddressAdd
- AddressInNaturallyAlignedBlock
- AddressIncrement
- ClearEventRegister
- ConditionHolds
- ConsumptionOfSpeculativeDataBarrier
- CurrentInstrSet
- CurrentSecurityState
- DSBAlias
- EL0
- EL2Enabled
- ELFromM32
- ELFromSPSR
- ELStateUsingAArch32
- ELStateUsingAArch32K
- ELUsingAArch32
- ELUsingAArch32K
- EffectiveEA
- EffectiveHCR_EL2_NVx
- EffectiveSCR_EL3_NS
- EffectiveSCR_EL3_RW
- EffectiveTGE
- EndOfInstruction
- EndOfInstructionException
- EndOfInstructionReason
- EnterLowPowerState
- EventRegister
- ExceptionalOccurrenceTargetState
- ExecuteAsNOP
- FIQPending
- GetAccumulatedFPEExceptions
- GetPSRFromPSTATE
- HaveAArch32
- HaveAArch32EL
- HaveAArch64
- HaveEL
- HaveSecureState
- HighestEL
- Hint_Yield

- [IRQPending](#)
- [IllegalExceptionReturn](#)
- [InstrSet](#)
- [InstructionFetchBarrier](#)
- [InstructionSynchronizationBarrier](#)
- [IsASEInstruction](#)
- [IsCurrentSecurityState](#)
- [IsEventRegisterSet](#)
- [IsHighestEL](#)
- [IsSecureBelowEL3](#)
- [IsWFEsleep](#)
- [IsWFIsleep](#)
- [Mode](#)
- [NonSecureOnlyImplementation](#)
- [PLOfEL](#)
- [PSTATE](#)
- [PhysicalCountInt](#)
- [PrivilegeLevel](#)
- [RestoredITBits](#)
- [SecureOnlyImplementation](#)
- [SecurityState](#)
- [SecurityStateAtEL](#)
- [SendEvent](#)
- [SendEventLocal](#)
- [SetAccumulatedFPEExceptions](#)
- [SetPSTATEFromPSR](#)
- [ShouldAdvanceHS](#)
- [ShouldAdvanceIT](#)
- [ShouldAdvanceSS](#)
- [SynchronizeContext](#)
- [TakeUnmaskedPhysicalSErrorInterrupts](#)
- [TakeUnmaskedSErrorInterrupts](#)
- [ThisInstr](#)
- [ThisInstrLength](#)
- [UsingAArch32](#)
- [ValidSecurityStateAtEL](#)
- [VirtualFIQPending](#)
- [VirtualIRQPending](#)
- [WFXType](#)
- [WaitForEvent](#)
- [WaitForInterrupt](#)
- [WatchpointRelatedSyndrome](#)
- [tlbi](#)
- [Broadcast](#)
- [BroadcastTLBI](#)
- [TLBI](#)
- [TLBILevel](#)

- [TLBIMemAttr](#)
- [TLBIOp](#)
- [TLBIRecord](#)
- [VMID](#)
- [Maximum](#)
- [Unbounded](#)
- [ConstrainUnpredictable](#)
- [ConstrainUnpredictableBits](#)
- [ConstrainUnpredictableBool](#)
- [ConstrainUnpredictableInteger](#)
- [ConstrainUnpredictableProcedure](#)
- [Constraint](#)
- [Unpredictable](#)
- [AdvSIMDEExpandImm](#)
- [PolynomialMult](#)
- [SatQ](#)
- [ShiftSat](#)
- [SignedSat](#)
- [SignedSatQ](#)
- [UnsignedRSqrtEstimate](#)
- [UnsignedRecipEstimate](#)
- [UnsignedSat](#)
- [UnsignedSatQ](#)

G1.2.3.1 EncodeLDFSC

```
// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

func EncodeLDFSC(statuscode : Fault, level : integer) => bits(6)
begin
    var result : bits(6);

    case statuscode of
        when Fault_Permission => result = '001100';
        when Fault_Translation => result = '000100';
        when Fault_AsyncExternal => result = '010001';
        when Fault_SyncExternal => result = '010000';
        when Fault_SyncExternalOnWalk => result = '0101'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_SyncParity => result = '011000';
        when Fault_SyncParityOnWalk => result = '0111'::level[1:0]; assert level IN {0,1,2,3};
        when Fault_AsyncParity => result = '011001';
        when Fault_Alignment => result = '100001';
        when Fault_Debug => result = '100010';
        when Fault_TLBConflict => result = '110000';
        when Fault_Lockdown => result = '110100'; // IMPLEMENTATION DEFINED
        when Fault_Exclusive => result = '110101'; // IMPLEMENTATION DEFINED
        otherwise => unreachable;
    end;

    return result;
end;
```

G1.2.3.2 IPAValid

```
// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

func IPAValid(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault_None;

    if fault.s2fslwalk then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Permission,
            Fault_Translation,
            Fault_AddressSize
        };
    elseif fault.secondstage then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Translation,
            Fault_AddressSize
        };
    else
        return FALSE;
    end;
end;
```

G1.2.3.3 IsAsyncAbort

```
// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

func IsAsyncAbort(statuscode : Fault) => boolean
begin
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});
end;

// IsAsyncAbort()
// =====

func IsAsyncAbort(fault : FaultRecord) => boolean
begin
    return IsAsyncAbort(fault.statuscode);
end;
```

G1.2.3.4 IsDebugException

```
// IsDebugException()
// =====

func IsDebugException(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
end;
```

G1.2.3.5 IsExternalAbort

```
// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

readonly func IsExternalAbort(statuscode : Fault) => boolean
begin
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk,
        Fault_AsyncExternal,
        Fault_AsyncParity
    });
end;

// IsExternalAbort()
// =====

readonly func IsExternalAbort(fault : FaultRecord) => boolean
begin
    return IsExternalAbort(fault.statuscode);
end;
```

G1.2.3.6 IsExternalAbortOnWalk

```
// IsExternalAbortOnWalk()
// =====

func IsExternalAbortOnWalk(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault_None;
    return fault.statuscode IN {Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk};
end;
```

G1.2.3.7 IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

readonly func IsExternalSyncAbort(statuscode : Fault) => boolean
begin
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk
    });
end;

// IsExternalSyncAbort()
// =====

readonly func IsExternalSyncAbort(fault : FaultRecord) => boolean
begin
    return IsExternalSyncAbort(fault.statuscode);
end;
```

G1.2.3.8 IsFault

```
// IsFault()
// =====
// Return TRUE if a fault is associated with an address descriptor

func IsFault(addrdesc : AddressDescriptor) => boolean
begin
    return addrdesc.fault.statuscode != Fault_None;
end;

// IsFault()
// =====
// Return TRUE if a fault is associated with a memory access.

func IsFault(fault : Fault) => boolean
begin
    return fault != Fault_None;
end;

// IsFault()
// =====
// Return TRUE if a fault is associated with status returned by memory.

func IsFault(retstatus : PhysMemRetStatus) => boolean
begin
    return retstatus.statuscode != Fault_None;
end;
```

G1.2.3.9 IsSErrorInterrupt

```
// IsSErrorInterrupt()
// =====
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

func IsSErrorInterrupt(statuscode : Fault) => boolean
begin
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});
end;

// IsSErrorInterrupt()
// =====

func IsSErrorInterrupt(fault : FaultRecord) => boolean
begin
    return IsSErrorInterrupt(fault.statuscode);
end;

// Add a specific type of return value for FaultSyndrome
type IssType of record {
    iss : bits(25),
    iss2 : bits(24)
};
```

G1.2.3.10 IsSecondStage

```
// IsSecondStage()
// =====

func IsSecondStage(fault : FaultRecord) => boolean
begin
    assert fault.statuscode != Fault_None;
```

```
        return fault.secondstage;  
    end;
```

G1.2.3.11 LSInstructionSyndrome

```
// LSInstructionSyndrome()  
// =====  
// Returns the extended syndrome information for a second stage fault.  
// [10] - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.  
// [9:8] - Access size.  
// [7] - Sign extended (for loads).  
// [6:2] - Transfer register.  
// [1] - Transfer register is 64-bit.  
// [0] - Instruction has acquire/release semantics.  
  
impdef func LSInstructionSyndrome() => bits(11)  
begin  
    return ARBITRARY : bits(11);  
end;
```

G1.2.3.12 CACHE_OP

```
// CACHE_OP()  
// =====  
// Performs Cache maintenance operations as per CacheRecord.  
  
impdef func CACHE_OP(cache : CacheRecord)  
begin  
    return;  
end;
```

G1.2.3.13 CPASAtPAS

```
// CPASAtPAS()  
// =====  
// Get cache PA space for given PA space.  
  
func CPASAtPAS(pas : PASpace) => CachePASpace  
begin  
    case pas of  
        when PAS_NonSecure =>  
            return CPAS_NonSecure;  
        when PAS_Secure =>  
            return CPAS_Secure;  
        otherwise =>  
            unreachable;  
    end;  
end;
```

G1.2.3.14 CPASAtSecurityState

```
// CPASAtSecurityState()  
// =====  
// Get cache PA space for given security state.  
  
func CPASAtSecurityState(ss : SecurityState) => CachePASpace  
begin  
    case ss of  
        when SS_NonSecure =>  
            return CPAS_NonSecure;  
        when SS_Secure =>  
            return CPAS_SecureNonSecure;  
    end;  
end;
```

G1.2.3.15 CacheRecord

```
// CacheRecord
// =====
// Details related to a cache operation.

type CacheRecord of record {
    acctype      : AccessType,           // Access type
    cacheop      : CacheOp,               // Cache operation
    opscope      : CacheOpScope,         // Cache operation type
    cachetype    : CacheType,           // Cache type
    regval       : bits(64),
    paddress     : FullAddress,
    vaddress     : bits(64),             // For VA operations
    setnum       : integer,             // For SW operations
    waynum       : integer,             // For SW operations
    level        : integer,             // For SW operations
    shareability : Shareability,
    is_vmid_valid : boolean,            // is vmid valid for current context
    vmid         : bits(NUM_VMIDBITS),
    is_asid_valid : boolean,            // is asid valid for current context
    asid         : bits(NUM_ASIDBITS),
    security     : SecurityState,
    // For cache operations to full cache or by setnum/waynum
    // For operations by address, PA space in paddress
    cpas : CachePASpace
};
```

G1.2.3.16 DecodeSW

```
// DecodeSW()
// =====
// Decode input value into setnum, waynum and level for SW instructions.

func DecodeSW(regval : bits(64), cachetype : CacheType) => (integer, integer, integer)
begin
    let level : integer = UInt(regval[3:1]);
    let (numsets, associativity, linesize) = GetCacheInfo(level, cachetype);
    // For the given level and cachetype, get the number of sets, associativity and
    // cache line size in terms of actual bytes.

    let waybits : integer{} = CeilLog2(associativity) as integer{0..10};
    let setbits : integer{} = CeilLog2(numsets) as integer{0..15};
    let linebits : integer{} = FloorLog2(linesize) as integer{4..11};

    let waynum : integer = if associativity == 1 then 0 else UInt(regval[31:32-waybits]);
    let setnum : integer = if numsets == 1 then 0 else UInt(regval[linebits +: setbits]);

    return (setnum, waynum, level);
end;
```

G1.2.3.17 GetCacheInfo

```
// GetCacheInfo()
// =====
// Returns numsets, associativity & linesize in terms of actual bytes.

impdef func GetCacheInfo(level : integer, cachetype : CacheType) => (integer, integer, integer)
begin
    let numsets : integer = ImpDefInt("Numsets for DC SW instructions");
    let associativity : integer = ImpDefInt("Associativity for DC SW instructions");
    let linesize : integer = ImpDefInt("Linesize for DC SW instructions");
    return (numsets, associativity, linesize);
end;
```

G1.2.3.18 IsZeroBit

```
// IsZeroBit()
// =====

func IsZeroBit{N}(x : bits(N)) => bit
begin
    return if IsZero(x) then '1' else '0';
end;
```

G1.2.3.19 NormalizeReal

```
// NormalizeReal
// =====
// Normalizes x to the form 1.xxx... x 2^y and returns (mantissa, exponent)

func NormalizeReal(x : real) => (real, integer)
begin
    let exponent : integer = ILog2(x);
    let mantissa : real = x / (2.0 ^ exponent);
    return (mantissa, exponent);
end;
```

G1.2.3.20 RShr

```
// RShr()
// =====
// Shift integer value right with rounding

func RShr(value : integer, shift : integer, round : boolean) => integer
begin
    assert shift > 0;
    if round then
        return (value + (1 << (shift - 1))) >> shift;
    else
        return value >> shift;
    end;
end;
```

G1.2.3.21 Reverse

```
// Reverse()
// =====
// Reverse subwords of M bits in an N-bit word

func Reverse{N}(word : bits(N), M : integer) => bits(N)
begin
    assert N MOD M == 0;
    var result : bits(N);
    let swsize : integer{} = M as integer{1..N};
    let sw : integer = N DIV swsize;
    for s = 0 to sw-1 do
        result[((sw - 1) - s)*:swsize] = word[s*:swsize];
    end;
    return result;
end;
```

G1.2.3.22 Signal

```
// Signal
// =====
// Available signal types

type Signal of enumeration {LOW, HIGH};
```

G1.2.3.23 counters

```

var PhysicalCount : bits(64);

var IsLocalTimeoutEventPending : boolean;
var LocalTimeoutVal : bits(64);           // Value to compare against the Virtual Counter Timer
                                           // to generate the local timeout event.

```

G1.2.3.24 AArch32_CheckTimerConditions

```

// AArch32_CheckTimerConditions()
// =====
// Checking timer conditions for all A32 timer registers

func AArch32_CheckTimerConditions()
begin
    var status : boolean;
    var offset : bits(64);
    offset = Zeros{64};
    assert !HaveAArch64();
    if CNTP_CTL().ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTP_CVAL(),
                                      CNTP_CTL().IMASK, InterruptID_CNTP);
        CNTP_CTL().ISTATUS = if status then '1' else '0';
    end;
    if HaveEL(EL2) && CNTHP_CTL().ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTHP_CVAL(),
                                      CNTHP_CTL().IMASK, InterruptID_CNTHP);
        CNTHP_CTL().ISTATUS = if status then '1' else '0';
    end;
    if CNTV_CTL().ENABLE == '1' then
        status = IsTimerConditionMet(CNTVOFF(), CNTV_CVAL(),
                                      CNTV_CTL().IMASK, InterruptID_CNTV);
        CNTV_CTL().ISTATUS = if status then '1' else '0';
    end;
    return;
end;

```

G1.2.3.25 CNTHCTL_EL2_VHE

```

// CNTHCTL_EL2_VHE()
// =====
// In the case where EL2 accesses the CNTHCTL_EL1 register, and the access
// is redirected to CNTHCTL_EL2 as a result of HCR_EL2.E2H being 1,
// then the bits of CNTHCTL_EL2 that are RES0 in CNTHCTL_EL1 are
// treated as being UNKNOWN. This function applies the UNKNOWN behavior.

func CNTHCTL_EL2_VHE(original_value : bits(64)) => bits(64)
begin
    assert PSTATE.EL == EL2;

    var return_value : bits(64) = original_value;
    return_value[19:18] = ARBITRARY : bits(2);
    return_value[16:10] = ARBITRARY : bits(7);
    return return_value;
end;

```

G1.2.3.26 GenericCounterTick

```

// GenericCounterTick()
// =====
// Increments PhysicalCount value for every clock tick.

func GenericCounterTick()
begin

```

```

var prev_physical_count : bits(64);
if CNTCR().EN == '0' then
    AArch32_CheckTimerConditions();
    return;
end;
prev_physical_count = PhysicalCountInt();
PhysicalCount[63:0] = PhysicalCount[63:0] + 1;
AArch32_CheckTimerConditions();
TestEventCNTP(prev_physical_count, PhysicalCountInt());
TestEventCNTV(prev_physical_count, PhysicalCountInt());
return;
end;

```

G1.2.3.27 IsTimerConditionMet

```

// IsTimerConditionMet()
// =====

func IsTimerConditionMet(offset : bits(64), compare_value : bits(64),
                        imask : bits(1), intid : InterruptID) => boolean
begin
    var condition_met : boolean;
    var level : Signal;
    condition_met = (UInt(PhysicalCountInt() - offset) - UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then HIGH else LOW;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
end;

```

G1.2.3.28 SetEventRegister

```

// SetEventRegister()
// =====
// Sets the Event Register of this PE

func SetEventRegister()
begin
    EventRegister = '1';
    return;
end;

```

G1.2.3.29 TestEventCNTP

```

// TestEventCNTP()
// =====
// Generate Event stream from the physical counter

func TestEventCNTP(prev_physical_count : bits(64), current_physical_count : bits(64))
begin
    var n : integer;
    if CNTHCTL().EVNTEN == '1' then
        n = UInt(CNTHCTL().EVNTI);
        if CNTHCTL().EVNTDIR == '0' then
            if prev_physical_count[n] == '0' && current_physical_count[n] == '1' then
                SetEventRegister();
            end;
        else
            if prev_physical_count[n] == '1' && current_physical_count[n] == '0' then
                SetEventRegister();
            end;
        end;
    end;
    return;
end;

```

G1.2.3.30 TestEventCNTV

```
// TestEventCNTV()
// =====
// Generate Event stream from the virtual counter

func TestEventCNTV(prev_physical_count : bits(64), current_physical_count : bits(64))
begin
    var offset : bits(64);
    var samplebit, previousbit : bit;
    var n : integer;
    if CNTKCTL().EVNTEN == '1' then
        n = UInt(CNTKCTL().EVNTI);
        offset = if HaveEL(EL2) then CNTVOFF() else Zeros{64};
        samplebit = (current_physical_count - offset)[n];
        previousbit = (prev_physical_count - offset)[n];
        if CNTKCTL().EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister(); end;
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister(); end;
        end;
    end;
    return;
end;
```

G1.2.3.31 BitReverse

```
// BitReverse()
// =====

func BitReverse{N}(data : bits(N)) => bits(N)
begin
    var result : bits(N);
    for i = 0 to N-1 do
        result[(N-i)-1] = data[i];
    end;
    return result;
end;
```

G1.2.3.32 Poly32Mod2

```
// Poly32Mod2()
// =====

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

func Poly32Mod2{N}(data_in : bits(N), poly : bits(32)) => bits(32)
begin
    assert N > 32;
    var data : bits(N) = data_in;
    for i = N-1 downto 32 do
        if data[i] == '1' then
            data[i-1:0] = data[i-1:0] XOR (poly::Zeros{i-32});
        end;
    end;
    return data[31:0];
end;
```

G1.2.3.33 AESInvMixColumns

```
// AESInvMixColumns()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

func AESInvMixColumns(op : bits (128)) => bits(128)
```

```

begin
    let in0 : bits(4*8) = op[ 96+:8] :: op[ 64+:8] :: op[ 32+:8] :: op[  0+:8];
    let in1 : bits(4*8) = op[104+:8] :: op[ 72+:8] :: op[ 40+:8] :: op[  8+:8];
    let in2 : bits(4*8) = op[112+:8] :: op[ 80+:8] :: op[ 48+:8] :: op[ 16+:8];
    let in3 : bits(4*8) = op[120+:8] :: op[ 88+:8] :: op[ 56+:8] :: op[ 24+:8];

    var out0 : bits(4*8);
    var out1 : bits(4*8);
    var out2 : bits(4*8);
    var out3 : bits(4*8);

    for c = 0 to 3 do
        out0[c*8+:8] = (FFmul0E(in0[c*8+:8]) XOR FFmul0B(in1[c*8+:8]) XOR FFmul0D(in2[c*8+:8]) XOR
                        FFmul09(in3[c*8+:8]));
        out1[c*8+:8] = (FFmul09(in0[c*8+:8]) XOR FFmul0E(in1[c*8+:8]) XOR FFmul0B(in2[c*8+:8]) XOR
                        FFmul0D(in3[c*8+:8]));
        out2[c*8+:8] = (FFmul0D(in0[c*8+:8]) XOR FFmul09(in1[c*8+:8]) XOR FFmul0E(in2[c*8+:8]) XOR
                        FFmul0B(in3[c*8+:8]));
        out3[c*8+:8] = (FFmul0B(in0[c*8+:8]) XOR FFmul0D(in1[c*8+:8]) XOR FFmul09(in2[c*8+:8]) XOR
                        FFmul0E(in3[c*8+:8]));
    end;

    return (
        out3[3*8+:8] :: out2[3*8+:8] :: out1[3*8+:8] :: out0[3*8+:8] ::
        out3[2*8+:8] :: out2[2*8+:8] :: out1[2*8+:8] :: out0[2*8+:8] ::
        out3[1*8+:8] :: out2[1*8+:8] :: out1[1*8+:8] :: out0[1*8+:8] ::
        out3[0*8+:8] :: out2[0*8+:8] :: out1[0*8+:8] :: out0[0*8+:8]
    );
end;

```

G1.2.3.34 AESInvShiftRows

```

// AESInvShiftRows()
// =====
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

func AESInvShiftRows(op : bits(128)) => bits(128)
begin
    return (
        op[ 31: 24] :: op[ 55: 48] :: op[ 79: 72] :: op[103: 96] ::
        op[127:120] :: op[ 23: 16] :: op[ 47: 40] :: op[ 71: 64] ::
        op[ 95: 88] :: op[119:112] :: op[ 15:  8] :: op[ 39: 32] ::
        op[ 63: 56] :: op[ 87: 80] :: op[111:104] :: op[  7:  0]
    );
end;

```

G1.2.3.35 AESInvSubBytes

```

// AESInvSubBytes()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

func AESInvSubBytes(op : bits(128)) => bits(128)
begin
    // Inverse S-box values
    let GF2_inv : bits(16*16*8) = (
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0          */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17[127:0] ::
        /*E*/ 0x619953833cbbbec8b0f52aae4d3be0a0[127:0] ::
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160[127:0] ::
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f[127:0] ::
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc[127:0] ::
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147[127:0] ::
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96[127:0] ::
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a[127:0] ::
    )
end;

```

```

/*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0[127:0] ::
/*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890[127:0] ::
/*5*/ 0x849d8da75746155edab9edfd5048706c[127:0] ::
/*4*/ 0x92b6655dcc5ca4d41698688664f6f872[127:0] ::
/*3*/ 0x25d18b6d49a25b76b224d92866a12e08[127:0] ::
/*2*/ 0x4ec3fa420b954cee3d23c2a632947b54[127:0] ::
/*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c[127:0] ::
/*0*/ 0xfbd7f3819ea340bf38a53630d56a0952[127:0]

);
var out : bits(128);
for i = 0 to 15 do
    out[i*8+:8] = GF2_inv[UInt(op[i*8+:8])*8+:8];
end;
return out;
end;

```

G1.2.3.36 AESMixColumns

```

// AESMixColumns()
// =====
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

func AESMixColumns(op : bits (128)) => bits(128)
begin
    let in0 : bits(4*8) = op[ 96+:8] :: op[ 64+:8] :: op[ 32+:8] :: op[  0+:8];
    let in1 : bits(4*8) = op[104+:8] :: op[ 72+:8] :: op[ 40+:8] :: op[  8+:8];
    let in2 : bits(4*8) = op[112+:8] :: op[ 80+:8] :: op[ 48+:8] :: op[ 16+:8];
    let in3 : bits(4*8) = op[120+:8] :: op[ 88+:8] :: op[ 56+:8] :: op[ 24+:8];

    var out0 : bits(4*8);
    var out1 : bits(4*8);
    var out2 : bits(4*8);
    var out3 : bits(4*8);

    for c = 0 to 3 do
        out0[c*8+:8] = (FFmul02(in0[c*8+:8]) XOR FFmul03(in1[c*8+:8]) XOR
                        in2[c*8+:8] XOR in3[c*8+:8]);
        out1[c*8+:8] = (FFmul02(in1[c*8+:8]) XOR FFmul03(in2[c*8+:8]) XOR
                        in3[c*8+:8] XOR in0[c*8+:8]);
        out2[c*8+:8] = (FFmul02(in2[c*8+:8]) XOR FFmul03(in3[c*8+:8]) XOR
                        in0[c*8+:8] XOR in1[c*8+:8]);
        out3[c*8+:8] = (FFmul02(in3[c*8+:8]) XOR FFmul03(in0[c*8+:8]) XOR
                        in1[c*8+:8] XOR in2[c*8+:8]);
    end;

    return (
        out3[3*8+:8] :: out2[3*8+:8] :: out1[3*8+:8] :: out0[3*8+:8] ::
        out3[2*8+:8] :: out2[2*8+:8] :: out1[2*8+:8] :: out0[2*8+:8] ::
        out3[1*8+:8] :: out2[1*8+:8] :: out1[1*8+:8] :: out0[1*8+:8] ::
        out3[0*8+:8] :: out2[0*8+:8] :: out1[0*8+:8] :: out0[0*8+:8]
    );
end;

```

G1.2.3.37 AESShiftRows

```

// AESShiftRows()
// =====
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

func AESShiftRows(op : bits(128)) => bits(128)
begin
    return (

```

```
        op[ 95: 88] :: op[ 55: 48] :: op[ 15:  8] :: op[103: 96] ::  
        op[ 63: 56] :: op[ 23: 16] :: op[111:104] :: op[ 71: 64] ::  
        op[ 31: 24] :: op[119:112] :: op[ 79: 72] :: op[ 39: 32] ::  
        op[127:120] :: op[ 87: 80] :: op[ 47: 40] :: op[  7:  0]  
    );  
end;
```

G1.2.3.38 AESSubBytes

```
// AESSubBytes()  
// =====  
// Transformation in the Cipher that processes the State using a nonlinear  
// byte substitution table (S-box) that operates on each of the State bytes  
// independently.  
  
func AESSubBytes(op : bits(128)) => bits(128)  
begin  
    // S-box values  
    let GF2 : bits(16*16*8) = (  
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */  
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c[127:0] ::  
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1[127:0] ::  
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70[127:0] ::  
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba[127:0] ::  
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7[127:0] ::  
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0[127:0] ::  
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160[127:0] ::  
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd[127:0] ::  
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351[127:0] ::  
        /*6*/ 0xa89f3c507f02f94585334d43fbaefd0[127:0] ::  
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153[127:0] ::  
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309[127:0] ::  
        /*3*/ 0x75b227ebe28012079a059618c323c704[127:0] ::  
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7[127:0] ::  
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca[127:0] ::  
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63[127:0]  
    );  
    var out : bits(128);  
    for i = 0 to 15 do  
        out[i*8+:8] = GF2[UInt(op[i*8+:8])*8+:8];  
    end;  
    return out;  
end;
```

G1.2.3.39 FFMul02

```
// FFMul02()  
// =====  
  
func FFMul02(b : bits(8)) => bits(8)  
begin  
    let FFMul_02 : bits(256*8) = (  
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */  
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB[127:0] ::  
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB[127:0] ::  
        /*D*/ 0xA5A7A1A3ADAF9ABB5B7B1B3BDBFB9BB[127:0] ::  
        /*C*/ 0x858781838D8F898B959791939D9F999B[127:0] ::  
        /*B*/ 0x656761636D6F696B757771737D7F797B[127:0] ::  
        /*A*/ 0x454741434D4F494B555751535D5F595B[127:0] ::  
        /*9*/ 0x252721232D2F292B353731333D3F393B[127:0] ::  
        /*8*/ 0x050701030D0F090B151711131D1F191B[127:0] ::  
        /*7*/ 0xFEFCFAF8F6F4F2F0EEEECEAE8E6E4E2E0[127:0] ::  
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0[127:0] ::  
        /*5*/ 0xEBECBAB8B6B4B2B0AEACAA8A6A4A2A0[127:0] ::  
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280[127:0] ::  
    );  
    return b * FFMul_02;  
end;
```

```
/*3*/ 0x7E7C7A78767472706E6C6A6866646260[127:0] ::  
/*2*/ 0x5E5C5A58565452504E4C4A4846444240[127:0] ::  
/*1*/ 0x3E3C3A38363432302E2C2A2826242220[127:0] ::  
/*0*/ 0x1E1C1A18161412100E0C0A0806040200[127:0]  
);  
return FEmul_02[UInt(b)*8+:8];  
end;
```

G1.2.3.40 FEmul03

```
// FEmul03()  
// =====  
  
func FEmul03(b : bits(8)) => bits(8)  
begin  
    let FEmul_03 : bits(256*8) = (  
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0          */  
        /*F*/ 0x1A191C1F16151013020104070E0D080B[127:0] ::  
        /*E*/ 0x2A292C2F26252023323134373E3D383B[127:0] ::  
        /*D*/ 0x7A797C7F76757073626164676E6D686B[127:0] ::  
        /*C*/ 0x4A494C4F46454043525154575E5D585B[127:0] ::  
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB[127:0] ::  
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB[127:0] ::  
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB[127:0] ::  
        /*8*/ 0x8A898C8F86858083929194979E9D989B[127:0] ::  
        /*7*/ 0x818287848D8E8B88999A9F9C95969390[127:0] ::  
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0[127:0] ::  
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0[127:0] ::  
        /*4*/ 0xD1D2D7D4DDEDEBD8C9CACFCCC5C6C3C0[127:0] ::  
        /*3*/ 0x414247444D4E4B48595A5F5C55565350[127:0] ::  
        /*2*/ 0x717277747D7E7B78696A6F6C65666360[127:0] ::  
        /*1*/ 0x212227242D2E2B28393A3F3C35363330[127:0] ::  
        /*0*/ 0x111217141D1E1B18090A0F0C05060300[127:0]  
    );  
    return FEmul_03[UInt(b)*8+:8];  
end;
```

G1.2.3.41 FEmul09

```
// FEmul09()  
// =====  
  
func FEmul09(b : bits(8)) => bits(8)  
begin  
    let FEmul_09 : bits(256*8) = (  
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0          */  
        /*F*/ 0x464F545D626B70790E071C152A233831[127:0] ::  
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1[127:0] ::  
        /*D*/ 0x7D746F6659504B42353C272E1118030A[127:0] ::  
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A[127:0] ::  
        /*B*/ 0x3039222B141D060F78716A635C554E47[127:0] ::  
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7[127:0] ::  
        /*9*/ 0x0B0219102F263D34434A5158676E757C[127:0] ::  
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC[127:0] ::  
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD[127:0] ::  
        /*6*/ 0x3A3328211E170C05727B6069565F444D[127:0] ::  
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6[127:0] ::  
        /*4*/ 0x0108131A252C373E49405B526D647F76[127:0] ::  
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB[127:0] ::  
        /*2*/ 0x4C455E5768617A73040D161F2029323B[127:0] ::  
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990[127:0] ::  
        /*0*/ 0x777E656C535A41483F362D241B120900[127:0]  
    );  
    return FEmul_09[UInt(b)*8+:8];  
end;
```

G1.2.3.42 FFMul0B

```
// FFMul0B()
// =====

func FFMul0B(b : bits(8)) => bits(8)
begin
    let FFMul_0B : bits(256*8) = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA[127:0] ::
        /*E*/ 0x1318050E3F3429224B405D56676C717A[127:0] ::
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1[127:0] ::
        /*C*/ 0x68637E75444F5259303B262D1C170A01[127:0] ::
        /*B*/ 0x555E434879726F640D061B10212A373C[127:0] ::
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C[127:0] ::
        /*9*/ 0x2E2538330209141F767D606B5A514C47[127:0] ::
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7[127:0] ::
        /*7*/ 0x545F424978736E650C071A11202B363D[127:0] ::
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D[127:0] ::
        /*5*/ 0x2F2439320308151E777C616A5B504D46[127:0] ::
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6[127:0] ::
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB[127:0] ::
        /*2*/ 0x1219040F3E3528234A415C57666D707B[127:0] ::
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0[127:0] ::
        /*0*/ 0x69627F74454E5358313A272C1D160B00[127:0]
    );
    return FFMul_0B[UInt(b)*8+:8];
end;
```

G1.2.3.43 FFMul0D

```
// FFMul0D()
// =====

func FFMul0D(b : bits(8)) => bits(8)
begin
    let FFMul_0D : bits(256*8) = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC[127:0] ::
        /*E*/ 0x474A5D50737E69642F2235381B16010C[127:0] ::
        /*D*/ 0x2C21363B1815020F44495E53707D6A67[127:0] ::
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7[127:0] ::
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1[127:0] ::
        /*A*/ 0x2A27303D1E130409424F5855767B6C61[127:0] ::
        /*9*/ 0x414C5B5675786F622924333E1D10070A[127:0] ::
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA[127:0] ::
        /*7*/ 0x4D40575A7974636E25283F32111C0B06[127:0] ::
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6[127:0] ::
        /*5*/ 0xF6BECE1C2CFD8D59E938489AAA7B0BD[127:0] ::
        /*4*/ 0x262B3C31121F08054E4354597A77606D[127:0] ::
        /*3*/ 0x202D3A3714190E034845525F7C71666B[127:0] ::
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB[127:0] ::
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0[127:0] ::
        /*0*/ 0x4B46515C7F726568232E3934171A0D00[127:0]
    );
    return FFMul_0D[UInt(b)*8+:8];
end;
```

G1.2.3.44 FFMul0E

```
// FFMul0E()
// =====

func FFMul0E(b : bits(8)) => bits(8)
begin
    let FFMul_0E : bits(256*8) = (
```

```

/*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
/*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7[127:0] ::
/*E*/ 0x6D63717F555B49471D13010F252B3937[127:0] ::
/*D*/ 0x56584A446E60727C26283A341E10020C[127:0] ::
/*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC[127:0] ::
/*B*/ 0x202E3C321816040A505E4C426866747A[127:0] ::
/*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A[127:0] ::
/*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1[127:0] ::
/*8*/ 0x1B150709232D3F316B657779535D4F41[127:0] ::
/*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896[127:0] ::
/*6*/ 0x2C22303E141A08065C52404E646A7876[127:0] ::
/*5*/ 0x17190B052F21333D67697B755F51434D[127:0] ::
/*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD[127:0] ::
/*3*/ 0x616F7D735957454B111F0D032927353B[127:0] ::
/*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB[127:0] ::
/*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0[127:0] ::
/*0*/ 0x5A544648626C7E702A243638121C0E00[127:0]

);
return FEmul_0E[UInt(b)*8+:8];
end;

```

G1.2.3.45 SHA256hash

```

// SHA256hash()
// =====

func SHA256hash(x_in : bits(128), y_in : bits(128), w : bits(128),
                part1 : boolean) => bits(128)
begin
    var chs, maj, t : bits(32);
    var x : bits(128) = x_in;
    var y : bits(128) = y_in;

    for e = 0 to 3 do
        chs = SHAchoose(y[31:0], y[63:32], y[95:64]);
        maj = SHAmajority{32}(x[31:0], x[63:32], x[95:64]);
        t = y[127:96] + SHA256hashSIGMA1(y[31:0]) + chs + w[e*:32];
        x[127:96] = t + x[127:96];
        y[127:96] = t + SHA256hashSIGMA0(x[31:0]) + maj;
        let yx : bits(256) = ROL(y :: x, 32);
        (y, x) = (yx[128+:128], yx[0+:128]);
    end;
    return (if part1 then x else y);
end;

```

G1.2.3.46 SHA256hashSIGMA0

```

// SHA256hashSIGMA0()
// =====

func SHA256hashSIGMA0(x : bits(32)) => bits(32)
begin
    return ROR(x, 2) XOR ROR(x, 13) XOR ROR(x, 22);
end;

```

G1.2.3.47 SHA256hashSIGMA1

```

// SHA256hashSIGMA1()
// =====

func SHA256hashSIGMA1(x : bits(32)) => bits(32)
begin
    return ROR(x, 6) XOR ROR(x, 11) XOR ROR(x, 25);
end;

```

G1.2.3.48 SHAchoose

```
// SHAchoose()
// =====

func SHAchoose(x : bits(32), y : bits(32), z : bits(32)) => bits(32)
begin
    return ((y XOR z) AND x) XOR z;
end;
```

G1.2.3.49 SHAmajority

```
// SHAmajority()
// =====

func SHAmajority{N}(x : bits(N), y : bits(N), z : bits(N)) => bits(N)
begin
    assert N IN {32, 64};
    return ((x AND y) OR ((x OR y) AND z));
end;
```

G1.2.3.50 SHAparity

```
// SHAparity()
// =====

func SHAparity(x : bits(32), y : bits(32), z : bits(32)) => bits(32)
begin
    return (x XOR y XOR z);
end;
```

G1.2.3.51 DecodeType

```
// DecodeType
// =====

type DecodeType of enumeration {
    Decode_UNDEF,
    Decode_NOP,
    Decode_OK
};
```

G1.2.3.52 EndOfDecode

```
// EndOfDecode()
// =====

// This function is invoked to end the Decode phase and performs Branch target Checks
// before taking any UNDEFINED exceptions, NOPs, or continuing to execute.

func EndOfDecode(reason : DecodeType)
begin
    case reason of
        when Decode_NOP => ExecuteAsNOP();
        when Decode_UNDEF => Undefined();
        when Decode_OK => pass; // Continue to execute.
    end;
    return;
end;
```

G1.2.3.53 ClearExclusiveByAddress

```
// ClearExclusiveByAddress()
// =====
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.

impdef func ClearExclusiveByAddress(paddress : FullAddress,
                                   processorid : integer,
                                   size : integer)

begin
    return;
end;
```

G1.2.3.54 ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====
// Clear the local Exclusives monitor for the specified processorid.

impdef func ClearExclusiveLocal(processorid : integer)

begin
    return;
end;
```

G1.2.3.55 ExclusiveMonitorsStatus

```
// ExclusiveMonitorsStatus()
// =====
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.

impdef func ExclusiveMonitorsStatus() => bit

begin
    return '0';
end;
```

G1.2.3.56 IsExclusiveGlobal

```
// IsExclusiveGlobal()
// =====
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

impdef func IsExclusiveGlobal(paddress : FullAddress,
                              processorid : integer,
                              size : integer) => boolean

begin
    return TRUE;
end;
```

G1.2.3.57 IsExclusiveLocal

```
// IsExclusiveLocal()
// =====
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

impdef func IsExclusiveLocal(paddress : FullAddress, processorid : integer,
```

```

                                size : integer) => boolean
begin
    return FALSE;
end;

```

G1.2.3.58 MarkExclusiveGlobal

```

// MarkExclusiveGlobal()
// =====
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.

impdef func MarkExclusiveGlobal(paddress : FullAddress,
                                processorid : integer,
                                size : integer)
begin
    return;
end;

```

G1.2.3.59 MarkExclusiveLocal

```

// MarkExclusiveLocal()
// =====
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.

impdef func MarkExclusiveLocal(paddress : FullAddress, processorid : integer, size : integer)
begin
    return;
end;

```

G1.2.3.60 ProcessorID

```

// ProcessorID()
// =====
// Return the ID of the currently executing PE.

impdef func ProcessorID() => integer
begin
    return UInt(MPIDR().Aff2 :: MPIDR().Aff1 :: MPIDR().Aff0);
end;

```

G1.2.3.61 HaveSoftwareLock

```

// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.

func HaveSoftwareLock(component : Component) => boolean
begin
    case component of
        when Component_Debug =>
            return ImpDefBool("Debug has Software Lock");
        when Component_PMU =>
            return ImpDefBool("PMU has Software Lock");
        when Component_CTI =>
            return ImpDefBool("CTI has Software Lock");
        otherwise =>
            unreachable;
    end;
end;

```

G1.2.3.62 HaveTraceExt

```
// HaveTraceExt()
// =====
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

readonly func HaveTraceExt() => boolean
begin
    return IsFeatureImplemented(FEAT_ETMv4);
end;
```

G1.2.3.63 ClearPendingPhysicalSError

```
// ClearPendingPhysicalSError()
// =====
// Clear a pending physical SError interrupt.

impdef func ClearPendingPhysicalSError()
begin
    return;
end;
```

G1.2.3.64 ClearPendingVirtualSError

```
// ClearPendingVirtualSError()
// =====
// Clear a pending virtual SError interrupt.

func ClearPendingVirtualSError()
begin
    HCR().VA = '0';
end;
```

G1.2.3.65 ExternalAbort

```
// ExternalAbort()
// =====

func ExternalAbort(fault : FaultRecord) recurselimit 1
begin
    if IsExternalSyncAbort(fault) then
        AArch32_Abort(fault);
    else
        PendSErrorInterrupt(fault);
    end;
end;
```

G1.2.3.66 ExternalFault

```
// ExternalFault()
// =====
// Return a fault recording indicating a fault for a Synchronous/Asynchronous External abort.

func ExternalFault(memretstatus : PhysMemRetStatus, iswrite : boolean,
    memaddrdesc : AddressDescriptor, size : integer,
    accdesc : AccessDescriptor) => FaultRecord
begin
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
        (memretstatus.statuscode IN {Fault_SyncParity, Fault_AsyncParity}));

    var fault : FaultRecord = NoFault(accdesc, memaddrdesc.vaddress);
    fault.statuscode = memretstatus.statuscode;
    fault.write      = iswrite;
```

```

    fault.extflag    = memretstatus.extflag;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                           size, accdesc)) then
        if fault.statuscode == Fault_SyncParity then
            fault.statuscode = Fault_AsyncParity;
        else
            fault.statuscode = Fault_AsyncExternal;
        end;
    end;

    return fault;
end;

```

G1.2.3.67 GetPendingPhysicalSError

```

// GetPendingPhysicalSError()
// =====
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

impdef func GetPendingPhysicalSError() => FaultRecord
begin
    return ARBITRARY : FaultRecord;
end;

```

G1.2.3.68 HandleExternalAbort

```

// HandleExternalAbort()
// =====
// Takes a Synchronous/Asynchronous abort based on fault.

func HandleExternalAbort(memretstatus : PhysMemRetStatus, iswrite : boolean,
                        memaddrdesc : AddressDescriptor, size : integer,
                        accdesc : AccessDescriptor)
begin
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
            (memretstatus.statuscode IN {Fault_SyncParity, Fault_AsyncParity}));

    let fault : FaultRecord = ExternalFault(memretstatus, iswrite, memaddrdesc, size, accdesc);
    ExternalAbort(fault);
end;

```

G1.2.3.69 HandleExternalReadAbort

```

// HandleExternalReadAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

func HandleExternalReadAbort(memstatus : PhysMemRetStatus, memaddrdesc : AddressDescriptor,
                            size : integer, accdesc : AccessDescriptor)
begin
    let iswrite : boolean = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
end;

```

G1.2.3.70 HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// =====
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

func HandleExternalTTWAbort(memretstatus : PhysMemRetStatus, iswrite : boolean,
                           memaddrdesc : AddressDescriptor,
                           accdesc : AccessDescriptor, size : integer,
                           input_fault : FaultRecord) => FaultRecord

begin
    var output_fault : FaultRecord = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                           size, accdesc)) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_AsyncParity;
        else
            output_fault.statuscode = Fault_AsyncExternal;
        end;
    end;

    // If a synchronous fault is on a translation table walk, then update the fault type.
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
        end;
    end;

    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    end;

    return output_fault;
end;
```

G1.2.3.71 HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

func HandleExternalWriteAbort(memstatus : PhysMemRetStatus, memaddrdesc : AddressDescriptor,
                              size : integer, accdesc : AccessDescriptor)

begin
    let iswrite : boolean = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
end;
```

G1.2.3.72 IsExternalAbortTakenSynchronously

```
// IsExternalAbortTakenSynchronously()
// =====
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
```

```
//  
// When FEAT_DoubleFault is not implemented, External aborts on instruction  
// fetches and translation table walks on instruction fetches can be either  
// synchronous or asynchronous.  
// When FEAT_DoubleFault is implemented, all External abort exceptions on  
// instruction fetches and translation table walks on instruction fetches  
// must be synchronous.  
  
impdef func IsExternalAbortTakenSynchronously(memstatus : PhysMemRetStatus, iswrite : boolean,  
                                              desc : AddressDescriptor, size : integer,  
                                              accdesc : AccessDescriptor) => boolean  
  
begin  
    return FALSE;  
end;
```

G1.2.3.73 IsPhysicalSErrorPending

```
// IsPhysicalSErrorPending()  
// =====  
// Returns TRUE if a physical SError interrupt is pending.  
  
impdef func IsPhysicalSErrorPending() => boolean  
begin  
    return FALSE;  
end;
```

G1.2.3.74 IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()  
// =====  
// Returns TRUE if the physical SError interrupt is edge-triggered  
// and FALSE otherwise.  
  
func IsSErrorEdgeTriggered() => boolean  
begin  
    return ImpDefBool("Edge-triggered SError");  
end;
```

G1.2.3.75 IsSynchronizablePhysicalSErrorPending

```
// IsSynchronizablePhysicalSErrorPending()  
// =====  
// Returns TRUE if a synchronizable physical SError interrupt is pending.  
  
impdef func IsSynchronizablePhysicalSErrorPending() => boolean  
begin  
    return FALSE;  
end;
```

G1.2.3.76 IsVirtualSErrorPending

```
// IsVirtualSErrorPending()  
// =====  
// Return TRUE if a virtual SError interrupt is pending.  
  
func IsVirtualSErrorPending() => boolean  
begin  
    return HCR().VA == '1';  
end;
```

G1.2.3.77 PendErrorInterrupt

```
// PendErrorInterrupt()
// =====
// Pend the SError Interrupt.

impdef func PendErrorInterrupt(fault : FaultRecord)
begin
    return;
end;
```

G1.2.3.78 FixedToFP

```
// FixedToFP()
// =====

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

func FixedToFP{N, M}(op : bits(M), fbits : integer, unsigned : boolean, fpcr : FPCR_Type,
    rounding : FPRounding) => bits(N)
begin
    assert N IN {32,64};
    assert M IN {16,32};
    var result : bits(N);
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    let int_operand : integer{} = if unsigned then UInt(op) else SInt(op);

    // Scale by fractional bits and generate a real value
    let real_operand : real = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero{N}('0');
    else
        result = FPRound{N}(real_operand, fpcr, rounding);
    end;

    return result;
end;
```

G1.2.3.79 FPAbs

```
// FPAbs()
// =====

func FPAbs{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    return '0' :: op[N-2:0];
end;
```

G1.2.3.80 FPAdd

```
// FPAdd()
// =====

func FPAdd{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin

    assert N IN {32,64};
    let rounding : FPRounding = FPRoundingMode(fpcr);
```

```

let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);

var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
if !done then
    let inf1 : boolean = (type1 == FType_Infinity);
    let inf2 : boolean = (type2 == FType_Infinity);
    let zero1 : boolean = (type1 == FType_Zero);
    let zero2 : boolean = (type2 == FType_Zero);
    if inf1 && inf2 && sign1 == NOT(sign2) then
        result = FPDefaultNaN{N}(fpcr);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
        result = FPInfinity{N}('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
        result = FPInfinity{N}('1');
    elsif zero1 && zero2 && sign1 == sign2 then
        result = FPZero{N}(sign1);
    else
        let result_value : real = value1 + value2;
        if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
            let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero{N}(result_sign);
        else
            result = FPRound{N}(result_value, fpcr, rounding);
        end;
    end;
end;

return result;
end;

```

G1.2.3.81 FPBits

```

// FPBits()
// =====
// Returns the minimum exponent, numbers of exponent and fraction bits.

func FPBits(N : integer) => FPBitsType
begin
    var F : FPFracBits;
    var minimum_exp : integer;
    if N == 16 then
        minimum_exp = -14; F = 10;
    elsif N == 32 then
        minimum_exp = -126; F = 23;
    else // N == 64
        minimum_exp = -1022; F = 52;
    end;

    return (F, minimum_exp);
end;

```

G1.2.3.82 FPBitsType

```

// FPBitsType
// =====

type FPBitsType of (FPFracBits, integer);

```

G1.2.3.83 FPFracBits

```
// FPFracBits
// =====

type FPFracBits of integer{2, 3, 7, 10, 23, 52};
```

G1.2.3.84 FPCompare

```
// FPCompare()
// =====

func FPCompare{N}(op1 : bits(N), op2 : bits(N), signal_nans : boolean, fpcr : FPCR_Type) => bits(4)
begin
    assert N IN {32,64};
    let (type1,sign1,value1) : (FType, bit, real) = FPUntpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FPUntpack{N}(op2, fpcr);

    var result : bits(4);
    if type1 IN {FType_SNaN, FType_QNaN} || type2 IN {FType_SNaN, FType_QNaN} then
        result = '0011';
        if type1 == FType_SNaN || type2 == FType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end;
    else
        // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';
        end;
    end;

    return result;
end;
```

G1.2.3.85 FPCompareEQ

```
// FPCompareEQ()
// =====

func FPCompareEQ{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {32,64};
    let (type1,sign1,value1) : (FType, bit, real) = FPUntpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FPUntpack{N}(op2, fpcr);

    var result : boolean;
    if type1 IN {FType_SNaN, FType_QNaN} || type2 IN {FType_SNaN, FType_QNaN} then
        result = FALSE;
        if type1 == FType_SNaN || type2 == FType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end;
    else
        // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        result = (value1 == value2);
    end;

    return result;
end;
```

G1.2.3.86 FPCmpareGE

```
// FPCmpareGE()
// =====

func FPCmpareGE{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {32,64};
    let (type1,sign1,value1) : (FType, bit, real) = FUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FUnpack{N}(op2, fpcr);

    var result : boolean;
    if type1 IN {FType_SNaN, FType_QNaN} || type2 IN {FType_SNaN, FType_QNaN} then
        result = FALSE;
        FProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FUnpack()
        result = (value1 >= value2);
    end;
    return result;
end;
```

G1.2.3.87 FPCmpareGT

```
// FPCmpareGT()
// =====

func FPCmpareGT{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => boolean
begin
    assert N IN {32,64};
    let (type1,sign1,value1) : (FType, bit, real) = FUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FUnpack{N}(op2, fpcr);

    var result : boolean;
    if type1 IN {FType_SNaN, FType_QNaN} || type2 IN {FType_SNaN, FType_QNaN} then
        result = FALSE;
        FProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FUnpack()
        result = (value1 > value2);
    end;
    return result;
end;
```

G1.2.3.88 FPConvert

```
// FPConvert()
// =====

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.

func FPConvert{M, N}(op : bits(N), fpcr : FPCR_Type, rounding : FRounding) => bits(M)
begin

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    var result : bits(M);

    // Unpack floating-point operand optionally with flush-to-zero.
    let (fptype,sign,value) = FUnpack{N}(op, fpcr);

    let alt_hp : boolean = (M == 16) && (fpcr.AHP == '1');

    if fptype == FType_SNaN || fptype == FType_QNaN then
        if alt_hp then
```

```

        result = FPZero{M}(sign);
    elsif fpcr.DN == '1' then
        result = FPDefaultNaN{M}(fpcr);
    else
        result = FPConvertNaN{M, N}(op);
    end;
    if fptype == FPType_SNaN || alt_hp then
        FPProcessException(FPExc_InvalidOp, fpcr);
    end;
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign::Ones{M-1};
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity{M}(sign);
        end;
    elsif fptype == FPType_Zero then
        result = FPZero{M}(sign);
    else
        result = FPRound{M}(value, fpcr, rounding);
    end;
    return result;
end;

// FPConvert()
// =====

func FPConvert{M, N}(op : bits(N), fpcr : FPCR_Type) => bits(M)
begin
    return FPConvert{M, N}(op, fpcr, FPRoundingMode(fpcr));
end;

```

G1.2.3.89 FPConvertNaN

```

// FPConvertNaN()
// =====
// Converts a NaN of one floating-point type to another

func FPConvertNaN{M, N}(op : bits(N)) => bits(M)
begin
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    var result : bits(M);
    var frac : bits(51);

    let sign : bit = op[N-1];

    // Unpack payload from input NaN
    case N of
        when 64 => frac = op[50:0];
        when 32 => frac = op[21:0]::Zeros{29};
        when 16 => frac = op[8:0]::Zeros{42};
    end;

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 => result = sign::Ones{M-52}::frac;
        when 32 => result = sign::Ones{M-23}::frac[50:29];
        when 16 => result = sign::Ones{M-10}::frac[50:42];
    end;

    return result;
end;

```

G1.2.3.90 FPDecodeRM

```
// FPDecodeRM()
// =====

// Decode most common AArch32 floating-point rounding encoding.

func FPDecodeRM(rm : bits(2)) => FPRounding
begin
    var result : FPRounding;
    case rm of
        when '00' => result = FPRounding_TIEAWAY; // A
        when '01' => result = FPRounding_TIEEVEN; // N
        when '10' => result = FPRounding_POSINF; // P
        when '11' => result = FPRounding_NEGINF; // M
    end;

    return result;
end;
```

G1.2.3.91 FPDecodeRounding

```
// FPDecodeRounding()
// =====

// Decode floating-point rounding mode and common AArch64 encoding.

func FPDecodeRounding(rmode : bits(2)) => FPRounding
begin
    case rmode of
        when '00' => return FPRounding_TIEEVEN; // N
        when '01' => return FPRounding_POSINF; // P
        when '10' => return FPRounding_NEGINF; // M
        when '11' => return FPRounding_ZERO; // Z
    end;
end;
```

G1.2.3.92 FPDefaultNaN

```
// FPDefaultNaN()
// =====

func FPDefaultNaN{N}(fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let sign : bit = '0';

    let exp : bits(E) = Ones{};
    let frac : bits(F) = '1'::Zeros{F-1};
    return sign :: exp :: frac;
end;
```

G1.2.3.93 FPDiv

```
// FPDiv()
// =====

func FPDiv{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    let (type1,sign1,value1) : (FPTType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPTType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
```

```

if !done then
    let inf1 : boolean = type1 == FPType_Infinity;
    let inf2 : boolean = type2 == FPType_Infinity;
    let zero1 : boolean = type1 == FPType_Zero;
    let zero2 : boolean = type2 == FPType_Zero;

    if (inf1 && inf2) || (zero1 && zero2) then
        result = FPDefaultNaN{N}(fpcr);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif inf1 || zero2 then
        result = FPInfinity{N}(sign1 XOR sign2);
        if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr); end;
    elsif zero1 || inf2 then
        result = FPZero{N}(sign1 XOR sign2);
    else
        result = FPRound{N}(value1/value2, fpcr);
    end;

end;
return result;
end;

```

G1.2.3.94 FPExc

```

// FPExc
// =====

type FPExc of enumeration {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                           FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

```

G1.2.3.95 FPInfinity

```

// FPInfinity()
// =====

func FPInfinity{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = Ones{};
    let frac : bits(F) = Zeros{};
    return sign :: exp :: frac;
end;

```

G1.2.3.96 FPMax

```

// FPMax()
// =====
// Compare two inputs and return the greater value after rounding. The
// 'fpcr' argument supplies the FPCR control bits.

func FPMax{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);

    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then

```



```

var fptype : FPType;
var sign : bit;
var value : real;
if value1 > value2 then
    (fptype,sign,value) = (type1,sign1,value1);
else
    (fptype,sign,value) = (type2,sign2,value2);
end;
if fptype == FPType_Infinity then
    result = FPInfinity{N}(sign);
elsif fptype == FPType_Zero then
    sign = sign1 AND sign2;          // Use most positive sign
    result = FPZero{N}(sign);
else
    // The use of FPRound() covers the case where there is a trapped underflow exception
    // for a denormalized number even though the result is exact.
    let rounding : FPRounding = FPRoundingMode(fpcr);
    result = FPRound{N}(value, fpcr, rounding);
end;
end;

return result;
end;

```

G1.2.3.97 FPMMaxNormal

```

// FPMMaxNormal()
// =====

func FPMMaxNormal{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = Ones{E-1}::'0';
    let frac : bits(F) = Ones{};
    return sign :: exp :: frac;
end;

```

G1.2.3.98 FPMMaxNum

```

// FPMMaxNum()
// =====

func FPMMaxNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    var op1 : bits(N) = op1_in;
    var op2 : bits(N) = op2_in;
    let (type1,-,-) = FPUnpack{N}(op1, fpcr);
    let (type2,-,-) = FPUnpack{N}(op2, fpcr);

    // Treat a single quiet-NaN as -Infinity.
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity{N}('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity{N}('1');
    end;
    let result : bits(N) = FPMMax{}(op1, op2, fpcr);

    return result;
end;

```

G1.2.3.99 IsMerging

```
// IsMerging()
// =====
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

func IsMerging(fpcr : FPCR_Type) => boolean
begin
    return FALSE;
end;
```

G1.2.3.100 FPMIn

```
// FPMIn()
// =====
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits.

func FPMIn{N}(op1 : bits(N), op2 : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    var done : boolean;
    var result : bits(N);
    var fpcr : FPCR_Type = fpcr_in;
    let (type1,sign1,value1) : (FType, bit, real) = FPUntpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FPUntpack{N}(op2, fpcr);

    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        var fptype : FType;
        var sign : bit;
        var value : real;
        var rounding : FPRounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        end;
        if fptype == FType_Infinity then
            result = FPinfinity{N}(sign);
        elsif fptype == FType_Zero then
            sign = sign1 OR sign2;          // Use most negative sign
            result = FPZero{N}(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            result = FPRound{N}(value, fpcr, rounding);
        end;
    end;
    return result;
end;
```

G1.2.3.101 FPMInNum

```
// FPMInNum()
// =====

func FPMInNum{N}(op1_in : bits(N), op2_in : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    var op1 : bits(N) = op1_in;
    var op2 : bits(N) = op2_in;
    let (type1,-,-) = FPUntpack{N}(op1, fpcr);
    let (type2,-,-) = FPUntpack{N}(op2, fpcr);
```

```
// Treat a single quiet-NaN as +Infinity.
if type1 == FPType_QNaN && type2 != FPType_QNaN then
    op1 = FPInfinity{N}('0');
elseif type1 != FPType_QNaN && type2 == FPType_QNaN then
    op2 = FPInfinity{N}('0');
end;
let result : bits(N) = FPMin{}(op1, op2, fpcr);

return result;
end;
```

G1.2.3.102 FPMul

```
// FPMul()
// =====

func FPMul{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);
    var (done,result) : (boolean, bits(N)) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean = (type1 == FPType_Infinity);
        let inf2 : boolean = (type2 == FPType_Infinity);
        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN{N}(fpcr);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elseif inf1 || inf2 then
            result = FPInfinity{N}(sign1 XOR sign2);
        elseif zero1 || zero2 then
            result = FPZero{N}(sign1 XOR sign2);
        else
            result = FPRound{N}(value1*value2, fpcr);
        end;
    end;

    return result;
end;
```

G1.2.3.103 FPMulAdd

```
// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits.

func FPMulAdd{N}(addend : bits(N), op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};

    let (typeA,signA,valueA) : (FPType, bit, real) = FPUnpack{N}(addend, fpcr);
    let (type1,sign1,value1) : (FPType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUnpack{N}(op2, fpcr);
    let rounding : FPRounding = FPRoundingMode(fpcr);
    let inf1 : boolean = (type1 == FPType_Infinity); let zero1 : boolean = (type1 == FPType_Zero);
    let inf2 : boolean = (type2 == FPType_Infinity); let zero2 : boolean = (type2 == FPType_Zero);

    var (done,result) : (boolean, bits(N)) = FPProcessNaNs3{N}(typeA, type1, type2,
                                                                addend, op1, op2, fpcr);
```

```

if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN{N}(fpcr);
    FPProcessException(FPExc_InvalidOp, fpcr);
end;

if !done then
    let infA : boolean = (typeA == FPType_Infinity);
    let zeroA : boolean = (typeA == FPType_Zero);

    // Determine sign and type product will have if it does not cause an
    // Invalid Operation.
    let signP : bit = sign1 XOR sign2;
    let infP : boolean = inf1 || inf2;
    let zeroP : boolean = zero1 || zero2;

    // Non SNaN-generated Invalid Operation cases are multiplies of zero
    // by infinity and additions of opposite-signed infinities.
    let invalidop : boolean = ((inf1 && zero2) || (zero1 && inf2) ||
                                (infA && infP && signA != signP));

    if invalidop then
        result = FPDefaultNaN{N}(fpcr);
        FPProcessException(FPExc_InvalidOp, fpcr);
    // Other cases involving infinities produce an infinity of the same sign.
    elsif (infA && signA == '0') || (infP && signP == '0') then
        result = FPInfinity{N}('0');
    elsif (infA && signA == '1') || (infP && signP == '1') then
        result = FPInfinity{N}('1');

    // Cases where the result is exactly zero and its sign is not determined by the
    // rounding mode are additions of same-signed zeros.
    elsif zeroA && zeroP && signA == signP then
        result = FPZero{N}(signA);

    // Otherwise calculate numerical result and round it.
    else
        let result_value : real = valueA + (value1 * value2);
        if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
            let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero{N}(result_sign);
        else
            result = FPRound{N}(result_value, fpcr);
        end;
    end;
end;
return result;
end;

```

G1.2.3.104 FPMuIX

```

// FPMuIX()
// =====

func FPMuIX{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    var result : bits(N);
    var done : boolean;
    let (type1,sign1,value1) : (FPType, bit, real) = FPUntpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FPType, bit, real) = FPUntpack{N}(op2, fpcr);

    (done,result) = FPProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean = (type1 == FPType_Infinity);
        let inf2 : boolean = (type2 == FPType_Infinity);

```

```

        let zero1 : boolean = (type1 == FPType_Zero);
        let zero2 : boolean = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo{N}(sign1 XOR sign2);
        elsif inf1 || inf2 then
            result = FPinfinity{N}(sign1 XOR sign2);
        elsif zero1 || zero2 then
            result = FPZero{N}(sign1 XOR sign2);
        else
            result = FPRound{N}(value1*value2, fpcr);
        end;

    end;

    return result;
end;
```

G1.2.3.105 FPNeg

```

// FPNeg()
// =====

func FPNeg{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    return NOT(op[N-1]) :: op[N-2:0];
end;
```

G1.2.3.106 FPOnePointFive

```

// FPOnePointFive()
// =====

func FPOnePointFive{N}(sign : bit) => bits(N)
begin
    assert N IN {32,64};
    let E : integer{} = (if N == 32 then 8 else 11);
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = '0'::Ones{E-1};
    let frac : bits(F) = '1'::Zeros{F-1};
    let result : bits((F + E) + 1) = sign :: exp :: frac;

    return result;
end;
```

G1.2.3.107 FPProcessException

```

// FPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

func FPProcessException(except : FPExc, fpcr : FPCR_Type)
begin
    var cumul : integer;
    // Determine the cumulative exception bit number
    case except of
        when FPExc_InvalidOp =>    cumul = 0;
        when FPExc_DivideByZero => cumul = 1;
        when FPExc_Overflow =>    cumul = 2;
        when FPExc_Underflow =>   cumul = 3;
        when FPExc_Inexact =>     cumul = 4;
        when FPExc_InputDenorm => cumul = 7;
```

```

end;
let enable : integer = cumul + 8;
if fpcr[enable] == '1' then
    // Trapping of the exception enabled.
    // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
    // and if so then how exceptions and in what order that they may be
    // accumulated before calling FPTrappedException().
    var accumulated_exceptions : bits(8) = GetAccumulatedFPExceptions();
    accumulated_exceptions[cumul] = '1';
    if ImpDefBool("Support trapping of floating-point exceptions") then
        if UsingAArch32() then
            AArch32_FPTrappedException(accumulated_exceptions);
        end;
    else
        // The exceptions generated by this instruction are accumulated by the PE and
        // FPTrappedException is called later during its execution, before the next
        // instruction is executed. This field is cleared at the start of each FP instruction.
        SetAccumulatedFPExceptions(accumulated_exceptions);
    end;
elseif UsingAArch32() then
    // Set the cumulative exception bit
    FPSCR()[cumul] = '1';
end;
return;
end;

```

G1.2.3.108 FPProcessNaN

```

// FPProcessNaN()
// =====
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.

func FPProcessNaN{N}(fptype : FType, op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    assert fptype IN {FType_QNaN, FType_SNaN};
    var topfrac : integer;

    case N of
        when 32 => topfrac = 22;
        when 64 => topfrac = 51;
    end;

    var result : bits(N) = op;
    if fptype == FType_SNaN then
        result[topfrac] = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    end;
    if fpcr.DN == '1' then // DefaultNaN requested
        result = FPDefaultNaN{N}(fpcr);
    end;
    return result;
end;

```

G1.2.3.109 FPProcessNaNs

```

// FPProcessNaNs()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is

```

```
// updated directly in the FPSR where appropriate.

func FPProcessNaNs{N}(type1 : FPType, type2 : FPType, op1 : bits(N),
                      op2 : bits(N), fpcr : FPCR_Type) => (boolean, bits(N))
begin
    assert N IN {32,64};
    var done : boolean;
    var result : bits(N);
    if type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN{N}(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN{N}(type2, op2, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN{N}(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN{N}(type2, op2, fpcr);
    else
        done = FALSE; result = Zeros{N}; // 'Don't care' result
    end;
    return (done, result);
end;
```

G1.2.3.110 FPProcessNaNs3

```
// FPProcessNaNs3()
// =====
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

func FPProcessNaNs3{N}(type1 : FPType, type2 : FPType, type3 : FPType,
                      op1 : bits(N), op2 : bits(N), op3 : bits(N),
                      fpcr : FPCR_Type) => (boolean, bits(N))
begin
    assert N IN {32,64};
    var result : bits(N);
    var done : boolean;

    if type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN{N}(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN{N}(type2, op2, fpcr);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN{N}(type3, op3, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN{N}(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN{N}(type2, op2, fpcr);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN{N}(type3, op3, fpcr);
    else
        done = FALSE; result = Zeros{N}; // 'Don't care' result
    end;
    return (done, result);
end;
```

G1.2.3.111 FPRecipEstimate

```
// FPRecipEstimate()
// =====
```

```

func FPRecipEstimate{N}(operand : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    var fpcr : FPCR_Type = fpcr_in;
    var result : bits(N);
    var overflow_to_inf : boolean;
    let (fptype,sign,value) : (FType, bit, real) = FPUnpack{N}(operand, fpcr);

    let rounding : FPRounding = FPRoundingMode(fpcr);
    if fptype == FType_SNaN || fptype == FType_QNaN then
        result = FPProcessNaN{N}(fptype, operand, fpcr);
    elsif fptype == FType_Infinity then
        result = FPZero{N}(sign);
    elsif fptype == FType_Zero then
        result = FPInfinity{N}(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
        (N == 32 && Abs(value) < 2.0^-128) ||
        (N == 64 && Abs(value) < 2.0^-1024)
    ) then
        case rounding of
            when FPRounding_TIEEVEN =>
                overflow_to_inf = TRUE;
            when FPRounding_POSINF =>
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF =>
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO =>
                overflow_to_inf = FALSE;
        end;
        result = if overflow_to_inf then FPInfinity{N}(sign) else FPMMaxNormal{N}(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif fpcr.FZ == '1'
        && (
            (N == 32 && Abs(value) >= 2.0^126) ||
            (N == 64 && Abs(value) >= 2.0^1022)
        ) then
        // Result flushed to zero of correct sign
        result = FPZero{N}(sign);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR().UFC = '1';
        end;
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        var fraction : bits(52);
        var exp : integer;
        case N of
            when 32 =>
                fraction = operand[22:0] :: Zeros{29};
                exp = UInt(operand[30:23]);
            when 64 =>
                fraction = operand[51:0];
                exp = UInt(operand[62:52]);
        end;

        if exp == 0 then
            if fraction[51] == '0' then
                exp = -1;
                fraction = fraction[49:0]::'00';
            else

```



```

        fraction = fraction[50:0]::'0';
    end;
end;

let scaled : integer = UInt('1'::fraction[51:44]);

var result_exp : integer;
case N of
    when 32 => result_exp = 253 - exp; // In range 253-254 = -1 to 253+1 = 254
    when 64 => result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046
end;

// Scaled is in range 256 .. 511 representing a fixed-point number in range [0.5 .. 1.0].
let estimate : integer = RecipEstimate(scaled);

// Estimate is in the range 256 .. 511 representing a fixed-point
// result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.
fraction = estimate[7:0] :: Zeros{44};

if result_exp == 0 then
    fraction = '1' :: fraction[51:1];
elsif result_exp == -1 then
    fraction = '01' :: fraction[51:2];
    result_exp = 0;
end;

case N of
    when 32 => result = sign :: result_exp[N-25:0] :: fraction[51:29];
    when 64 => result = sign :: result_exp[N-54:0] :: fraction[51:0];
end;
end;

return result;
end;

```

G1.2.3.112 RecipEstimate

```

// RecipEstimate()
// =====
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range 1.0 to 511/256.

func RecipEstimate(a_in : integer) => integer
begin
    var a : integer = a_in;
    assert 256 <= a && a < 512;
    a = a*2+1; // Round to nearest
    let b : integer = (2 ^ 19) DIVRM a;
    let r : integer = (b+1) DIVRM 2; // Round to nearest
    assert 256 <= r && r < 512;
    return r;
end;

```

G1.2.3.113 FPrecpX

```

// FPrecpX()
// =====

func FPrecpX{N}(op : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};

```

```

var fpcr : FPCR_Type = fpcr_in;
let (F, -) = FPBits(N);
let E = (N - F) - 1;
var result : bits(N);
var exp : bits(E);
var max_exp : bits(E);
let frac : bits(F) = Zeros{};

let (fptype,sign,value) : (FType, bit, real) = FPUnpack{N}(op, fpcr);
exp = op[F+:E];
max_exp = Ones{E} - 1;

if fptype == FType_SNaN || fptype == FType_QNaN then
    result = FPProcessNaN{N}(fptype, op, fpcr);
else
    if IsZero(exp) then // Zero and denormals
        result = ZeroExtend{N}(sign::max_exp::frac);
    else // Infinities and normals
        result = ZeroExtend{N}(sign::NOT(exp)::frac);
    end;
end;

return result;
end;

```

G1.2.3.114 FPRound

```

// FPRound()
// =====
// Generic conversion from precise, unbounded real data type to IEEE format.

func FPRound{N}(op : real, fpcr : FPCR_Type) => bits(N)
begin
    return FPRound{N}(op, fpcr, FPRoundingMode(fpcr));
end;

// FPRound()
// =====
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'. The 'fpcr' argument supplies FPCR control
// bits. Status information is updated directly in the FPSR where appropriate.

func FPRound{N}(op : real, fpcr : FPCR_Type, rounding : FPRounding) => bits(N)
begin
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    var result : bits(N);

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    let (F, minimum_exp) = FPBits(N);
    let E = (N - F) - 1;
    // Split value into sign, unrounded mantissa and exponent.
    var sign : bit;
    var exponent : integer;
    var mantissa : real;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    end;
    (mantissa, exponent) = NormalizeReal(mantissa);

    if fpcr.FZ == '1' && N != 16 && exponent < minimum_exp then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then

```

```

        FPSCR().UFC = '1';
    end;
    return FPZero{N}(sign);
end;

// Start creating the exponent value for the result. Start by biasing the actual exponent
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
var biased_exp : integer = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent); end;

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
// < 2.0^F if biased_exp == 0, >= 2.0^F if not
var int_mant : integer = RoundDown(mantissa * 2.0^F);
var error : real = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped.
let trapped_UF : boolean = fpcr.UFE == '1';
if biased_exp == 0 && (error != 0.0 || trapped_UF) then
    FPProcessException(FPExc_Underflow, fpcr);
end;

// Round result according to rounding mode.
var round_up : boolean;
var overflow_to_inf : boolean;
case rounding of
    when FPRounding_TIEEVEN =>
        round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));
        overflow_to_inf = TRUE;
    when FPRounding_POSINF =>
        round_up = (error != 0.0 && sign == '0');
        overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF =>
        round_up = (error != 0.0 && sign == '1');
        overflow_to_inf = (sign == '1');
    when FPRounding_ZERO, FPRounding_ODD =>
        round_up = FALSE;
        overflow_to_inf = FALSE;
end;

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then // Rounded up from denormalized to normalized
        biased_exp = 1;
    end;
    if int_mant == 2^(F+1) then // Rounded up to next exponent
        biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;
    end;
end;

// Handle rounding to odd
if error != 0.0 && rounding == FPRounding_ODD && int_mant[0] == '0' then
    int_mant = int_mant + 1;
end;

// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half precision
    if biased_exp >= 2^E - 1 then
        result = if overflow_to_inf then FPInfinity{N}(sign) else FPMaxNormal{N}(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        error = 1.0; // Ensure that an Inexact exception occurs
    else
        result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0] :: Zeros{N-(E+F+1)};
    end;
else // Alternative half precision

```

```

        if biased_exp >= 2^E then
            result = sign :: Ones{N-1};
            FFPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0; // Ensure that an Inexact exception does not occur
        else
            result = sign :: biased_exp[E-1:0] :: int_mant[F-1:0] :: Zeros{N-(E+F+1)};
        end;
    end;

    // Deal with Inexact exception.
    if error != 0.0 then
        FFPProcessException(FPExc_Inexact, fpcr);
    end;

    return result;
end;

```

G1.2.3.115 FPRounding

```

// FPRounding
// =====
// The conversion and rounding functions take an explicit
// rounding mode enumeration instead of booleans or FPCR values.

type FPRounding of enumeration {FPRounding_TIEEVEN, FPRounding_POSINF,
                                FPRounding_NEGINF, FPRounding_ZERO,
                                FPRounding_TIEAWAY, FPRounding_ODD};

```

G1.2.3.116 FPRoundingMode

```

// FPRoundingMode()
// =====
// Return the current floating-point rounding mode.

func FPRoundingMode(fpcr : FPCR_Type) => FPRounding
begin
    return FPDecodeRounding(fpcr.RMode);
end;

```

G1.2.3.117 FPRoundInt

```

// FPRoundInt()
// =====

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

func FPRoundInt{N}(op : bits(N), fpcr : FPCR_Type, rounding : FPRounding,
                    exact : boolean) => bits(N)
begin
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    let (fptype,sign,value) : (FType, bit, real) = FPUnpack{N}(op, fpcr);

    var result : bits(N);
    if fptype == FType_SNaN || fptype == FType_QNaN then
        result = FFPProcessNaN{N}(fptype, op, fpcr);
    elsif fptype == FType_Infinity then
        result = FPInfinity{N}(sign);
    elsif fptype == FType_Zero then
        result = FPZero{N}(sign);
    else

```

```
// Extract integer component.
var int_result : integer = RoundDown(value);
let error : real = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment.
var round_up : boolean;
case rounding of
  when FPRounding_TIEEVEN =>
    round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
  when FPRounding_POSINF =>
    round_up = (error != 0.0);
  when FPRounding_NEGINF =>
    round_up = FALSE;
  when FPRounding_ZERO =>
    round_up = (error != 0.0 && int_result < 0);
  when FPRounding_TIEAWAY =>
    round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
end;

if round_up then int_result = int_result + 1; end;

// Convert integer value into an equivalent real value.
let real_result : real = Real(int_result);

// Re-encode as a floating-point value, result is always exact.
if real_result == 0.0 then
  result = FPZero{N}(sign);
else
  result = FPRound{N}(real_result, fpcr, FPRounding_ZERO);
end;

// Generate inexact exceptions.
if error != 0.0 && exact then
  FPProcessException(FPExc_Inexact, fpcr);
end;

end;

return result;
end;
```

G1.2.3.118 FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =====

func FPRSqrtEstimate{N}(operand : bits(N), fpcr_in : FPCR_Type) => bits(N)
begin
  assert N IN {32,64};
  var fpcr : FPCR_Type = fpcr_in;

  let (fptype,sign,value) : (FPTType, bit, real) = FPUnpack{N}(operand, fpcr);

  var result : bits(N);
  if fptype == FPTType_SNaN || fptype == FPTType_QNaN then
    result = FPProcessNaN{N}(fptype, operand, fpcr);
  elsif fptype == FPTType_Zero then
    result = FPInfinity{N}(sign);
    FPProcessException(FPExc_DivideByZero, fpcr);
  elsif sign == '1' then
    result = FPDefaultNaN{N}(fpcr);
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif fptype == FPTType_Infinity then
    result = FPZero{N}('0');
  else
    // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
    // evenness or oddness of the exponent unchanged, and calculate result exponent.
```

```
// Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
// biased version of -1 or -2, fraction = original fraction extended with zeros.

var fraction : bits(52);
var exp : integer;
case N of
    when 32 =>
        fraction = operand[22:0] :: Zeros{29};
        exp      = UInt(operand[30:23]);
    when 64 =>
        fraction = operand[51:0];
        exp      = UInt(operand[62:52]);
end;

if exp == 0 then
    while fraction[51] == '0' looplimit 51 do
        fraction = fraction[50:0] :: '0';
        exp      = exp - 1;
    end;
    fraction = fraction[50:0] :: '0';
end;

var scaled : integer;
if exp[0] == '0' then
    scaled = UInt('1'::fraction[51:44]);
else
    scaled = UInt('01'::fraction[51:45]);
end;

var result_exp : integer;
case N of
    when 32 => result_exp = ( 380 - exp) DIVRM 2;
    when 64 => result_exp = (3068 - exp) DIVRM 2;
end;

let estimate : integer = RecipSqrtEstimate(scaled);

// Estimate is in the range 256 .. 511 representing a fixed point
// result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit and high-order
// fraction bits, and exponent calculated above.
case N of
    when 32 => result = '0' :: result_exp[N-25:0] :: estimate[7:0]::Zeros{15};
    when 64 => result = '0' :: result_exp[N-54:0] :: estimate[7:0]::Zeros{44};
end;
end;

return result;
end;
```

G1.2.3.119 RecipSqrtEstimate

```
// RecipSqrtEstimate()
// =====
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

func RecipSqrtEstimate(a_in : integer) => integer
begin
    var a : integer = a_in;
    assert 128 <= a && a < 512;
    if a < 256 then // 0.25 .. 0.5
        a = a*2+1; // a in units of 1/512 rounded to nearest
    else // 0.5 .. 1.0
```

```

        a = (a >> 1) << 1;          // Discard bottom bit
        a = (a+1)*2;                // a in units of 1/256 rounded to nearest
    end;
    var b : integer = 512;
    while a*(b+1)*(b+1) < 2^28 looplimit 510 do
        b = b+1;                    // b = largest b such that b < 2^14 / sqrt(a)
    end;
    let r : integer = (b+1) DIVRM 2; // Round to nearest
    assert 256 <= r && r < 512;
    return r;
end;

```

G1.2.3.120 FPSqrt

```

// FPSqrt()
// =====

func FPSqrt{N}(op : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    var (fptype,sign,value) : (FType, bit, real) = FPUnpack{N}(op, fpcr);

    var result : bits(N);
    if fptype == FType_SNaN || fptype == FType_QNaN then
        result = FProcessNaN{N}(fptype, op, fpcr);
    elsif fptype == FType_Zero then
        result = FPZero{N}(sign);
    elsif fptype == FType_Infinity && sign == '0' then
        result = FPInfinity{N}(sign);
    elsif sign == '1' then
        result = FPDefaultNaN{N}(fpcr);
        FProcessException(FPExc_InvalidOp, fpcr);
    else
        var prec : integer;
        if N == 16 then
            prec = 13; // 10 fraction bits + 3
        elsif N == 32 then
            prec = 26; // 23 fraction bits + 3
        else // N == 64
            prec = 55; // 52 fraction bits + 3
        end;
        value = SqrtRounded(value, prec);
        result = FPRound{N}(value, fpcr);
    end;
    return result;
end;

```

G1.2.3.121 FPSub

```

// FPSub()
// =====

func FPSub{N}(op1 : bits(N), op2 : bits(N), fpcr : FPCR_Type) => bits(N)
begin
    assert N IN {32,64};
    let rounding : FRounding = FRoundingMode(fpcr);

    let (type1,sign1,value1) : (FType, bit, real) = FPUnpack{N}(op1, fpcr);
    let (type2,sign2,value2) : (FType, bit, real) = FPUnpack{N}(op2, fpcr);

    var (done,result) : (boolean, bits(N)) = FProcessNaNs{N}(type1, type2, op1, op2, fpcr);
    if !done then
        let inf1 : boolean = (type1 == FType_Infinity);
        let inf2 : boolean = (type2 == FType_Infinity);
    end;

```

```

let zero1 : boolean = (type1 == FType_Zero);
let zero2 : boolean = (type2 == FType_Zero);

if inf1 && inf2 && sign1 == sign2 then
    result = FPDefaultNaN{N}(fpcr);
    FProcessException(FPExc_InvalidOp, fpcr);
elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
    result = FPInfinity{N}('0');
elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
    result = FPInfinity{N}('1');
elseif zero1 && zero2 && sign1 == NOT(sign2) then
    result = FPZero{N}(sign1);
else
    let result_value : real = value1 - value2;
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        let result_sign : bit = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero{N}(result_sign);
    else
        result = FPRound{N}(result_value, fpcr, rounding);
    end;
end;

end;
return result;
end;

```

G1.2.3.122 FPThree

```

// FPThree()
// =====

func FPThree{N}(sign : bit) => bits(N)
begin
    assert N IN {32,64};
    let E : integer{} = (if N == 32 then 8 else 11);
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = '1'::Zeros{E-1};
    let frac : bits(F) = '1'::Zeros{F-1};
    let result : bits((F + E) + 1) = sign :: exp :: frac;

    return result;
end;

```

G1.2.3.123 FPToFixed

```

// FPToFixed()
// =====

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

func FPToFixed{M, N}(op : bits(N), fbits : integer, unsigned : boolean, fpcr : FPCR_Type,
    rounding : FPRounding) => bits(M)
begin
    assert N IN {32,64};
    assert M IN {16,32};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    var (fptype, sign, value) : (FType, bit, real) = FUnpack{N}(op, fpcr);

    // If NaN, set cumulative flag or take exception.
    if fptype == FType_SNaN || fptype == FType_QNaN then
        FProcessException(FPExc_InvalidOp, fpcr);
    end;
end;

```



```

end;

// Scale by fractional bits and produce integer rounded towards minus-infinity.
value = value * 2.0^fbits;
var int_result : integer = RoundDown(value);
let error : real      = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment.
var round_up : boolean;
case rounding of
  when FPRounding_TIEEVEN =>
    round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
  when FPRounding_POSINF =>
    round_up = (error != 0.0);
  when FPRounding_NEGINF =>
    round_up = FALSE;
  when FPRounding_ZERO =>
    round_up = (error != 0.0 && int_result < 0);
  when FPRounding_TIEAWAY =>
    round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
end;

if round_up then int_result = int_result + 1; end;

// Generate saturated result and exceptions.
let (result, overflow) : (bits(M), boolean) = SatQ{M}(int_result, unsigned);
if overflow then
  FPProcessException(FPExc_InvalidOp, fpcr);
elsif error != 0.0 then
  FPProcessException(FPExc_Inexact, fpcr);
end;

return result;
end;

```

G1.2.3.124 FPTwo

```

// FPTwo()
// =====

func FPTwo{N}(sign : bit) => bits(N)
begin
  assert N IN {32,64};
  let E : integer{} = (if N == 32 then 8 else 11);
  let F : integer{} = N - (E + 1);
  let exp : bits(E) = '1'::Zeros{E-1};
  let frac : bits(F) = Zeros{};
  let result : bits((F + E) + 1) = sign :: exp :: frac;
  return result;
end;

```

G1.2.3.125 FPType

```

// FPType
// =====

type FPType of enumeration {FPType_Zero,
                           FPType_Denormal,
                           FPType_Nonzero,
                           FPType_Infinity,
                           FPType_QNaN,
                           FPType_SNaN};

```

G1.2.3.126 FPUnpack

```

// FPUnpack()
// =====
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

func FPUnpack{N}(fpval : bits(N), fpcr_in : FPCR_Type) => (FType, bit, real)
begin
    assert N IN {16,32,64};
    let fpcr : FPCR_Type = fpcr_in;

    var value : real;
    var sign : bit;
    var fptype : FType;

    if N == 16 then
        sign = fpval[15];
        let exp16 : bits(5) = fpval[14:10];
        let frac16 : bits(10) = fpval[9:0];
        if IsZero(exp16) then
            // Produce zero if value is zero
            if IsZero(frac16) then
                fptype = FType_Zero; value = 0.0;
            else
                fptype = FType_Denormal; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
            end;
        elsif IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FType_Infinity; value = 2.0^1000000;
            else
                fptype = if frac16[9] == '1' then FType_QNaN else FType_SNaN;
                value = 0.0;
            end;
        else
            fptype = FType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);
        end;

    elsif N == 32 then
        sign = fpval[31];
        let exp32 : bits(8) = fpval[30:23];
        let frac32 : bits(23) = fpval[22:0];

        if IsZero(exp32) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac32) || fpcr.FZ == '1' then
                fptype = FType_Zero; value = 0.0;
            if !IsZero(frac32) then // Denormalized input flushed to zero
                FProcessException(FPExc_InputDenorm, fpcr);
            end;
        else
            fptype = FType_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        end;
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                fptype = FType_Infinity; value = 2.0^1000000;
            else
                fptype = if frac32[22] == '1' then FType_QNaN else FType_SNaN;
                value = 0.0;
            end;
        end;
    end;
end;

```

```

        end;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
    end;

else // N == 64
    sign = fpval[63];
    let exp64 : bits(11) = fpval[62:52];
    let frac64 : bits(52) = fpval[51:0];

    if IsZero(exp64) then
        // Produce zero if value is zero or flush-to-zero is selected.
        if IsZero(frac64) || fpcr.FZ == '1' then
            fptype = FPType_Zero; value = 0.0;
            if !IsZero(frac64) then // Denormalized input flushed to zero
                FPProcessException(FPExc_InputDenorm, fpcr);
            end;
        else
            fptype = FPType_Denormal; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        end;
    elseif IsOnes(exp64) then
        if IsZero(frac64) then
            fptype = FPType_Infinity; value = 2.0^1000000;
        else
            fptype = if frac64[51] == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
        end;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
    end;
end;

if sign == '1' then value = -value; end;

return (fptype, sign, value);
end;

```

G1.2.3.127 FPZero

```

// FPZero()
// =====

func FPZero{N}(sign : bit) => bits(N)
begin
    assert N IN {16,32,64};
    let E : integer{} = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    let F : integer{} = N - (E + 1);
    let exp : bits(E) = Zeros{};
    let frac : bits(F) = Zeros{};
    let result : bits((F + E) + 1) = sign :: exp :: frac;
    return result;
end;

```

G1.2.3.128 VFPEExpandImm

```

// VFPEExpandImm()
// =====

func VFPEExpandImm{N}(imm8 : bits(8)) => bits(N)
begin
    assert N IN {32,64};
    let E : integer{} = (if N == 32 then 8 else 11);
    let F : integer{} = (N - E) - 1;

```

```

    let sign : bit = imm8[7];
    let exp : bits(E) = NOT(imm8[6])::Replicate{E - 3}(imm8[6])::imm8[5:4];
    let frac : bits(F) = imm8[3:0]::Zeros{F-4};
    let result : bits((F + E) + 1) = sign :: exp :: frac;

    return result;
end;

```

G1.2.3.129 AddWithCarry

```

// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

func AddWithCarry{N}(x : bits(N), y : bits(N), carry_in : bit) => (bits(N), bits(4))
begin
    let unsigned_sum : integer = UInt(x) + UInt(y) + UInt(carry_in);
    let signed_sum : integer = SInt(x) + SInt(y) + UInt(carry_in);
    let result : bits(N) = unsigned_sum[N-1:0]; // same value as signed_sum[N-1:0]
    let n : bit = result[N-1];
    let z : bit = if IsZero(result) then '1' else '0';
    let c : bit = if UInt(result) == unsigned_sum then '0' else '1';
    let v : bit = if SInt(result) == signed_sum then '0' else '1';
    return (result, n::z::c::v);
end;

```

G1.2.3.130 InterruptID

```

// InterruptID
// =====

type InterruptID of enumeration {
    InterruptID_PMUIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,
};

```

G1.2.3.131 SetInterruptRequestLevel

```

// SetInterruptRequestLevel()
// =====
// Set a level-sensitive interrupt to the specified level.

impdef func SetInterruptRequestLevel(id : InterruptID, level : Signal)
begin
    return;
end;

```

G1.2.3.132 AccessDescriptor

```
// AccessDescriptor
// =====
// Memory access or translation invocation details that steer architectural behavior

type AccessDescriptor of record {
    acctype : AccessType,
    el : bits(2),                // Acting EL for the access
    ss : SecurityState,          // Acting Security State for the access
    acqsc : boolean,             // Acquire with Sequential Consistency
    relsc : boolean,             // Release with Sequential Consistency
    exclusive : boolean,         // Access has Exclusive semantics
    nontemporal : boolean,       // Hints the access is non-temporal
    read : boolean,              // Read from memory or only require read permissions
    write : boolean,             // Write to memory or only require write permissions
    cacheop : CacheOp,           // DC/IC: Cache operation
    opscope : CacheOpScope,      // DC/IC: Scope of cache operation
    cachetype : CacheType,       // DC/IC: Type of target cache
    a32lsmd : boolean,           // A32 Load/Store Multiple Data access
    within8byte : boolean,       // R32 - Checks if access is within 8 byte boundary
                                // for Load/Store Multiple Data access

    Rt : integer,                // Register named Rt in the instruction
    Rt2 : integer,               // Register named Rt2 in the instruction
    Rs : integer,                // Register named Rs in the instruction
    Rs2 : integer,               // Register named Rs2 in the instruction
    ispair : boolean,            // Access represents a Load/Store pair access
    lowestaddress : boolean,     // Is the current access the lowest address accessed by
                                // this instruction
};
```

G1.2.3.133 AccessType

```
// AccessType
// =====

type AccessType of enumeration {
    AccessType_IFETCH,          // Instruction FETCH
    AccessType_GPR,             // Software load/store to a General Purpose Register
    AccessType_ASIMD,           // Software ASIMD extension load/store instructions
    AccessType_IC,              // Sysop IC
    AccessType_DC,              // Sysop DC (not DC {Z,G,GZ}VA)
    AccessType_DCZero,          // Sysop DC {Z,G,GZ}VA
    AccessType_AT,              // Sysop AT
    AccessType_TTW              // Translation Table Walk
};
```

G1.2.3.134 AddrTop

```
// AddrTop()
// =====
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

func AddrTop(address : bits(64), el : bits(2)) => AddressSize
begin
    assert HaveEL(el);
    return 31;
end;
```

G1.2.3.135 AddressSize

```
// AddressSize
// =====

type AddressSize of integer{3..64};
```

G1.2.3.136 AlignmentEnforced

```
// AlignmentEnforced()
// =====
// For the active translation regime, determine if alignment is required by all accesses

func AlignmentEnforced() => boolean
begin
    let regime : Regime = TranslationRegime(PSTATE.EL);
    var A : bit;
    case regime of
        when Regime_EL2 => A = HSCTLR().A;
        when Regime_EL10 => A = SCTLR().A;
        otherwise => unreachable;
    end;
    return A == '1';
end;
```

G1.2.3.137 AllInAlignedQuantity

```
// AllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

readonly func AllInAlignedQuantity(N : integer{32, 64})(address : bits(N),
                                                         size : integer,
                                                         alignment : integer) => boolean

begin
    return (AlignDownSize(address+(size-1), alignment as AddressSize) ==
            AlignDownSize(address, alignment as AddressSize));
end;
```

G1.2.3.138 Allocation

```
// Allocation hints
// =====

constant MemHint_No : bits(2) = '00';    // No Read-Allocate, No Write-Allocate
constant MemHint_WA : bits(2) = '01';    // No Read-Allocate, Write-Allocate
constant MemHint_RA : bits(2) = '10';    // Read-Allocate, No Write-Allocate
constant MemHint_RWA : bits(2) = '11';   // Read-Allocate, Write-Allocate
```

G1.2.3.139 BigEndian

```
// BigEndian()
// =====

func BigEndian(acctype : AccessType) => boolean
begin
    var bigend : boolean;

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    end;
    return bigend;
end;
```

G1.2.3.140 BigEndianReverse

```
// BigEndianReverse()  
// =====  
  
func BigEndianReverse{width}(value : bits(width)) => bits(width)  
begin  
    assert width IN {8, 16, 32, 64, 128, 256};  
  
    if width == 8 then return value; end;  
    return Reverse{width}(value, 8);  
end;
```

G1.2.3.141 CacheOp

```
// CacheOp  
// =====  
  
type CacheOp of enumeration {  
    CacheOp_Clean,  
    CacheOp_Invalidate,  
    CacheOp_CleanInvalidate  
};
```

G1.2.3.142 CacheOpScope

```
// CacheOpScope  
// =====  
  
type CacheOpScope of enumeration {  
    CacheOpScope_SetWay,  
    CacheOpScope_PoU,  
    CacheOpScope_PoC,  
    CacheOpScope_ALLU,  
    CacheOpScope_ALLUIS  
};
```

G1.2.3.143 CachePASpace

```
// CachePASpace  
// =====  
  
type CachePASpace of enumeration {  
    CPAS_NonSecure,  
    CPAS_SecureNonSecure,    // Applicable only for DC *SW / IC IALLU* in Secure state:  
                             // match entries from Secure or Non-Secure PAS  
    CPAS_Secure  
};
```

G1.2.3.144 CacheType

```
// CacheType  
// =====  
  
type CacheType of enumeration {  
    CacheType_Data,  
    CacheType_Instruction  
};
```

G1.2.3.145 Cacheability

```
// Cacheability attributes
// =====

constant MemAttr_NC : bits(2) = '00';    // Non-cacheable
constant MemAttr_WT : bits(2) = '10';    // Write-through
constant MemAttr_WB : bits(2) = '11';    // Write-back
```

G1.2.3.146 CreateAccDescA32LSMD

```
// CreateAccDescA32LSMD()
// =====
// Access descriptor for A32 loads/store multiple general purpose registers

func CreateAccDescA32LSMD(memop : MemOp, within8byte : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.a32lsmd       = TRUE;
    accdesc.within8byte    = within8byte;

    return accdesc;
end;
```

G1.2.3.147 CreateAccDescASIMD

```
// CreateAccDescASIMD()
// =====
// Access descriptor for ASIMD&FP loads/stores

func CreateAccDescASIMD(memop : MemOp, nontemporal : boolean,
                        privileged : boolean) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    return CreateAccDescASIMD(memop, nontemporal, privileged, ispair);
end;

// CreateAccDescASIMD()
// =====

func CreateAccDescASIMD(memop : MemOp, nontemporal : boolean, privileged : boolean,
                        ispair : boolean) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_ASIMD);

    accdesc.nontemporal    = nontemporal;
    accdesc.el             = if !privileged then EL0 else PSTATE.EL;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.ispair         = ispair;
    return accdesc;
end;
```

G1.2.3.148 CreateAccDescAT

```
// CreateAccDescAT()
// =====
// Access descriptor for address translation operations

func CreateAccDescAT(ss : SecurityState, el : bits(2), ataccess : ATAccess) => AccessDescriptor
begin
```



```

var accdesc : AccessDescriptor = NewAccDesc(AccessType_AT);

accdesc.el          = el;
accdesc.ss          = ss;
case ataccess of
    when ATAccess_Read =>
        accdesc.(read, write) = (TRUE, FALSE);
    when ATAccess_Write =>
        accdesc.(read, write) = (FALSE, TRUE);
end;

return accdesc;
end;

```

G1.2.3.149 CreateAccDescAcqRel

```

// CreateAccDescAcqRel()
// =====
// Access descriptor for general purpose register loads/stores with ordering semantics

func CreateAccDescAcqRel(memop : MemOp, acqsc : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescAcqRel(memop, acqsc, Rt);
end;

func CreateAccDescAcqRel(memop : MemOp, acqsc : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescAcqRel(memop, ispair, acqsc, Rt, Rt2);
end;

func CreateAccDescAcqRel(memop : MemOp, ispair : boolean, acqsc : boolean, Rt : integer,
                        Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.acqsc          = acqsc;
    accdesc.reisc          = memop == MemOp_STORE;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.ispair         = ispair;
    accdesc.Rt             = Rt;
    accdesc.Rt2            = Rt2;

    return accdesc;
end;

```

G1.2.3.150 CreateAccDescDC

```

// CreateAccDescDC()
// =====
// Access descriptor for data cache operations

func CreateAccDescDC(cache : CacheRecord) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_DC);

    accdesc.cacheop        = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope        = cache.opscope;

    return accdesc;
end;

```

G1.2.3.151 CreateAccDescDCZero

```
// CreateAccDescDCZero()
// =====
// Access descriptor for data cache zero operations

func CreateAccDescDCZero() => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_DCZero);

    accdesc.write          = TRUE;

    return accdesc;
end;
```

G1.2.3.152 CreateAccDescExLDST

```
// CreateAccDescExLDST()
// =====
// Access descriptor for general purpose register loads/stores with exclusive semantics

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean, privileged : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescExLDST(memop, acqrel, privileged, Rt);
end;
```

```
// CreateAccDescExLDST()
// =====

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean,
                        privileged : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescExLDST(memop, acqrel, privileged, ispair, Rt, Rt2);
end;
```

```
// CreateAccDescExLDST()
// =====

func CreateAccDescExLDST(memop : MemOp, acqrel : boolean, privileged : boolean,
                        ispair : boolean, Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.acqsc          = acqrel && memop == MemOp_LOAD;
    accdesc.relsc          = acqrel && memop == MemOp_STORE;
    accdesc.exclusive      = TRUE;
    accdesc.el             = if !privileged then EL0 else PSTATE.EL;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.ispair         = ispair;
    accdesc.Rt             = Rt;
    accdesc.Rt2            = Rt2;

    return accdesc;
end;
```

G1.2.3.153 CreateAccDescGPR

```
// CreateAccDescGPR()
// =====
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics
```

```

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean,
                      privileged : boolean) => AccessDescriptor
begin
    let Rt : integer = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, Rt);
end;

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean,
                      privileged : boolean, Rt : integer) => AccessDescriptor
begin
    let ispair : boolean = FALSE;
    let Rt2 : integer = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, ispair, Rt, Rt2);
end;

func CreateAccDescGPR(memop : MemOp, nontemporal : boolean, privileged : boolean,
                      ispair : boolean, Rt : integer, Rt2 : integer) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_GPR);

    accdesc.el          = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal = nontemporal;
    accdesc.read         = memop == MemOp_LOAD;
    accdesc.write        = memop == MemOp_STORE;
    accdesc.Rt           = Rt;
    accdesc.Rt2          = Rt2;
    accdesc.ispair       = ispair;

    return accdesc;
end;

```

G1.2.3.154 CreateAccDescIC

```

// CreateAccDescIC()
// =====
// Access descriptor for instruction cache operations

func CreateAccDescIC(cache : CacheRecord) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_IC);

    accdesc.cacheop      = cache.cacheop;
    accdesc.cachetype    = cache.cachetype;
    accdesc.opscope      = cache.opscope;

    return accdesc;
end;

```

G1.2.3.155 CreateAccDescIFetch

```

// CreateAccDescIFetch()
// =====
// Access descriptor for instruction fetches

func CreateAccDescIFetch() => AccessDescriptor
begin
    let accdesc : AccessDescriptor = NewAccDesc(AccessType_IFETCH);

    return accdesc;
end;

```

G1.2.3.156 CreateAccDescS1TTW

```
// CreateAccDescS1TTW()
// =====
// Access descriptor for stage 1 translation table walks

func CreateAccDescS1TTW(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_TTW);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;

    return accdesc;
end;
```

G1.2.3.157 CreateAccDescS2TTW

```
// CreateAccDescS2TTW()
// =====
// Access descriptor for stage 2 translation table walks

func CreateAccDescS2TTW(accdesc_in : AccessDescriptor) => AccessDescriptor
begin
    var accdesc : AccessDescriptor = NewAccDesc(AccessType_TTW);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;

    return accdesc;
end;
```

G1.2.3.158 DataMemoryBarrier

```
// DataMemoryBarrier()
// =====

impdef func DataMemoryBarrier(types : MBReqTypes, vmid_sensitive : boolean)
begin
    return;
end;
```

G1.2.3.159 DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// =====

impdef func DataSynchronizationBarrier(scope : MBMaintenanceScope,
                                       types : MBReqTypes,
                                       vmid_sensitive : boolean)
begin
    return;
end;
```

G1.2.3.160 DeviceType

```
// DeviceType
// =====
// Extended memory types for Device memory.

type DeviceType of enumeration {DeviceType_GRE, DeviceType_nGRE,
                                DeviceType_nGnRE, DeviceType_nGnRnE};
```

G1.2.3.161 Fault

```
// Fault
// =====
// Fault types.

type Fault of enumeration {
    Fault_None,
    Fault_AccessFlag,
    Fault_Alignment,
    Fault_Background,
    Fault_Domain,
    Fault_Permission,
    Fault_Translation,
    Fault_AddressSize,
    Fault_SyncExternal,
    Fault_SyncExternalOnWalk,
    Fault_SyncParity,
    Fault_SyncParityOnWalk,
    Fault_AsyncParity,
    Fault_AsyncExternal,
    Fault_Debug,
    Fault_TLBConflict,
    Fault_Lockdown,
    Fault_Exclusive,
    Fault_ICacheMaint};
```

G1.2.3.162 FaultRecord

```
// FaultRecord
// =====
// Fields that relate only to Faults.

type FaultRecord of record {
    statuscode : Fault,           // Fault Status
    accessdesc : AccessDescriptor, // Details of the faulting access
    vaddress   : bits(64),        // Faulting virtual address
    ipaddress  : bits(NUM_PABITS), // Intermediate physical address
    s2fslwalk : boolean,         // Is on a Stage 1 translation table walk
    write      : boolean,         // TRUE for a write, FALSE for a read
    level      : integer,         // For translation, access flag and Permission faults
    extflag    : bit,            // IMPLEMENTATION DEFINED syndrome for External aborts
    secondstage : boolean,        // Is a Stage 2 abort
    domain     : bits(4),         // Domain number, AArch32 only
    watchptinfo : WatchpointInfo, // Watchpoint related fields
    debugmoe   : bits(4)         // Debug method of entry, from AArch32 only
};
```

G1.2.3.163 FullAddress

```
// FullAddress
// =====
// Physical or Intermediate Physical Address type.
// Although AArch32 only has access to 40 bits of physical or intermediate physical address space,
// the full address type has NUM_PABITS bits to allow interprocessing with AArch64.
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED,
// but never exceeds NUM_PABITS bits.

type FullAddress of record {
    paspace : PASpace,
    address : bits(NUM_PABITS)
};
```

G1.2.3.164 Hint_Prefetch

```
// Hint_Prefetch()
// =====
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

impdef func Hint_Prefetch(address : bits(64),
                        hint : PrefetchHint,
                        target : integer,
                        stream : boolean)

begin
    return;
end;
```

G1.2.3.165 IsWatchpointableAccess

```
// IsWatchpointableAccess()
// =====
// Return TRUE if access should be checked for watchpoints.

func IsWatchpointableAccess(accdesc : AccessDescriptor) => boolean
begin
    return (!(accdesc.acctype IN {AccessType_IFETCH,
                                AccessType_TTW,
                                AccessType_DC,
                                AccessType_IC,
                                AccessType_AT}) ||
            (accdesc.acctype == AccessType_DC &&
             accdesc.cacheop == CacheOp_Invalidate &&
             (!UsingAArch32() || (ImpDefBool("DCIMVAC generates watchpoint")))));
end;
```

G1.2.3.166 MBMaintenanceScope

```
// MBMaintenanceScope
// =====
// Memory barrier scope.

type MBMaintenanceScope of enumeration {MBMaintenanceScope_Nonshareable,
                                         MBMaintenanceScope_InnerShareable,
                                         MBMaintenanceScope_OuterShareable,
                                         MBMaintenanceScope_None};
```

G1.2.3.167 MBReqTypes

```
// MBReqTypes
// =====
// Memory barrier read/write.

type MBReqTypes of enumeration {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

G1.2.3.168 MPURecord

```
// MPURecord
// =====

type MPURecord of record {
    permissions : Permissions,
    memattrs : MemoryAttributes
};
```

G1.2.3.169 MemAttrHints

```
// MemAttrHints
// =====
// Attributes and hints for Normal memory.

type MemAttrHints of record {
    attrs : bits(2), // See MemAttr_*, Cacheability attributes
    hints : bits(2), // See MemHint_*, Allocation hints
    transient : boolean
};
```

G1.2.3.170 MemOp

```
// MemOp
// =====
// Memory access instruction types.

type MemOp of enumeration {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

G1.2.3.171 MemType

```
// MemType
// =====
// Basic memory types.

type MemType of enumeration {MemType_Normal, MemType_Device};
```

G1.2.3.172 MemoryAttributes

```
// MemoryAttributes
// =====
// Memory attributes descriptor

type MemoryAttributes of record {
    memtype : MemType,
    device : DeviceType, // For Device memory types
    inner : MemAttrHints, // Inner hints and attributes
    outer : MemAttrHints, // Outer hints and attributes
    shareability : Shareability, // Shareability attribute
};
```

G1.2.3.173 NewAccDesc

```
// NewAccDesc()
// =====
// Create a new AccessDescriptor with initialized fields

func NewAccDesc(acctype : AccessType) => AccessDescriptor
begin
    var accdesc : AccessDescriptor;

    accdesc.acctype = acctype;
    accdesc.el = PSTATE.EL;
```

```

    accdesc.ss                = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc             = FALSE;
    accdesc.rels             = FALSE;
    accdesc.exclusive         = FALSE;
    accdesc.nontemporal       = FALSE;
    accdesc.read              = FALSE;
    accdesc.write             = FALSE;
    accdesc.a32lsmd           = FALSE;
    accdesc.Rs                = -1;
    accdesc.Rs2               = -1;
    accdesc.Rt                = -1;
    accdesc.Rt2               = -1;
    accdesc.ispair            = FALSE;
    accdesc.lowestaddress     = TRUE;

    return accdesc;
end;

```

G1.2.3.174 PASpace

```

// PASpace
// =====
// Physical address spaces

type PASpace of enumeration {
    PAS_Secure,
    PAS_NonSecure
};

```

G1.2.3.175 Permissions

```

// Permissions
// =====
// Access Control bits in PMSAv32 registers.

type Permissions of record {
    ap_table : bits(2), // Stage 1 hierarchical access permissions
    xn_table : bit,     // Stage 1 hierarchical execute-never for single EL regimes
    pxn_table : bit,    // Stage 1 hierarchical privileged execute-never
    uxn_table : bit,    // Stage 1 hierarchical unprivileged execute-never
    ap : bits(3),       // Stage 1 access permissions
    xn : bit,           // Stage 1 execute-never for single EL regimes
    uxn : bit,          // Stage 1 unprivileged execute-never
    pxn : bit,          // Stage 1 privileged execute-never
    s2ap : bits(2),    // Stage 2 access permissions
    s2xn : bit         // Stage 2 execute-never
};

```

G1.2.3.176 PhysMemRead

```

// PhysMemRead()
// =====
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.

impdef func PhysMemRead(size : integer{8, 16, 32, 64, 128, 256, 512}){
    desc : AddressDescriptor,
    accdesc : AccessDescriptor
    => (PhysMemRetStatus, bits(size))

begin
    return (ARBITRARY : PhysMemRetStatus, ARBITRARY : bits(size));
end;

```

G1.2.3.177 PhysMemRetStatus

```
// PhysMemRetStatus
// =====
// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus of record {
    statuscode : Fault,          // Fault Status
    extflag : bit,              // IMPLEMENTATION DEFINED syndrome for External aborts
};
```

G1.2.3.178 PhysMemWrite

```
// PhysMemWrite()
// =====
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.

impdef func PhysMemWrite{size : integer{8, 16, 32, 64, 128, 256, 512}}(desc : AddressDescriptor,
                                                                    accdesc : AccessDescriptor,
                                                                    value : bits(size)) => PhysMemRetStatus

begin
    return ARBITRARY : PhysMemRetStatus;
end;
```

G1.2.3.179 PrefetchHint

```
// PrefetchHint
// =====
// Prefetch hint types.

type PrefetchHint of enumeration {
    Prefetch_READ,
    Prefetch_WRITE,
    Prefetch_EXEC
};
```

G1.2.3.180 S1AccessControls

```
// S1AccessControls
// =====
// Effective access controls defined by stage 1 translation

type S1AccessControls of record {
    r : bit,          // Stage 1 base read permission
    w : bit,          // Stage 1 base write permission
    x : bit,          // Stage 1 base execute permission
    wxn : bit         // Stage 1 write permission implies execute-never
};
```

G1.2.3.181 S2AccessControls

```
// S2AccessControls
// =====
// Effective access controls defined by stage 2 translation

type S2AccessControls of record {
    r : bit,          // Stage 2 read permission.
    w : bit,          // Stage 2 write permission.
    x : bit,          // Stage 2 execute permission.
};
```

G1.2.3.182 Shareability

```
// Shareability
// =====

type Shareability of enumeration {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

G1.2.3.183 SpeculativeStoreBypassBarrierToPA

```
// SpeculativeStoreBypassBarrierToPA()
// =====

impdef func SpeculativeStoreBypassBarrierToPA()
begin
    // Since there is no speculation in ASL model, this is a NOP.
    return;
end;
```

G1.2.3.184 SpeculativeStoreBypassBarrierToVA

```
// SpeculativeStoreBypassBarrierToVA()
// =====

impdef func SpeculativeStoreBypassBarrierToVA()
begin
    // Since there is no speculation in ASL model, this is a NOP.
    return;
end;
```

G1.2.3.185 VARange

```
// VARange
// =====
// Virtual address ranges

type VARange of enumeration {
    VARange_LOWER,
    VARange_UPPER
};
```

G1.2.3.186 ASID

```
// ASID
// ====
// Effective ASID.

func ASID() => bits(NUM_ASIDBITS)
begin
    return CONTEXTIDR().PROCID[(NUM_ASIDBITS-1):0];
end;
```

G1.2.3.187 BranchTo

```
// BranchTo()
// =====
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

func BranchTo{N}(target : bits(N), branch_type : BranchType, branch_conditional : boolean)
begin
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend{64}(target);
    end;
    return;
end;
```

G1.2.3.188 BranchToAddr

```
// BranchToAddr()
// =====
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

func BranchToAddr{N}(target : bits(N), branch_type : BranchType)
begin
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend{64}(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target[63:0];
    end;
    return;
end;
```

G1.2.3.189 BranchType

```
// BranchType
// =====
// Information associated with a change in control flow.

type BranchType of enumeration {
    BranchType_DIRCALL,    // Direct Branch with link
    BranchType_INDICAL,    // Indirect Branch with link
    BranchType_ERET,       // Exception return (indirect)
    BranchType_DBGEXIT,    // Exit from Debug state
    BranchType_RET,        // Indirect branch with function return hint
    BranchType_DIR,        // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};    // Other
```

G1.2.3.190 Branchtypetaken

```
// Branchtypetaken
// =====
var Branchtypetaken : BranchType;
```

G1.2.3.191 ESize

```
// ESize
// =====

type ESize of integer{8, 16, 32, 64, 128};
```

G1.2.3.192 EffectiveFPCR

```
// EffectiveFPCR()
// =====
// Returns the effective FPCR value

func EffectiveFPCR() => FPCR_Type
begin
    var fpcr : FPCR_Type = ZeroExtend{64}(FPSCR());
    fpcr[7:0]  = '00000000';
    fpcr[31:27] = '00000';
    return fpcr;
end;
```

G1.2.3.193 Hint_Branch

```
// Hint_Branch()
// =====
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.

impdef func Hint_Branch(hint : BranchType)
begin
    Branchtypetaken = hint;
    return;
end;
```

G1.2.3.194 NextInstrAddr

```
// NextInstrAddr()
// =====
// Return address of the sequentially next instruction.

impdef func NextInstrAddr{N}() => bits(N)
begin
    return Zeros{N};
end;
```

G1.2.3.195 ResetExternalDebugRegisters

```
// ResetExternalDebugRegisters()
// =====
// Reset the External Debug registers in the Core power domain.

impdef func ResetExternalDebugRegisters(cold_reset : boolean)
begin
    return;
end;
```

G1.2.3.196 ThisInstrAddr

```
// ThisInstrAddr()
// =====
// Return address of the current instruction.

func ThisInstrAddr{N}() => bits(N)
begin
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC[N-1:0];
end;
```

G1.2.3.197 UnimplementedIDRegister

```
// UnimplementedIDRegister()
// =====
// Trap access to unimplemented encodings in the feature ID register space.

func UnimplementedIDRegister()
begin
    Undefined();
end;
```

G1.2.3.198 V

```
// V - accessor
// =====

accessor V{width : ESize}(n : integer) <=> value : bits(width)
begin
    // Read from SIMD&FP register with implicit slice of 8, 16
    // 32, 64 or 128 bits.
    getter
        assert n >= 0 && n <= 31;
        return _V[[n]][width-1:0];
    end;
    // Write to SIMD&FP register with implicit extension from
    // 8, 16, 32, 64 or 128 bits.
    setter
        assert n >= 0 && n <= 31;
        _V[[n]] = ZeroExtend{128}(value);
    end;
end;
```

G1.2.3.199 Vpart

```
// Vpart - accessor
// =====

accessor Vpart{width : ESize}(n : integer, part : integer) <=> value : bits(width)
begin
    // Reads a 128-bit SIMD&FP register in up to two parts:
    // part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
    // part 1 returns only the top half of the 128-bit value held in the register.
    getter
        assert n >= 0 && n <= 31;
        assert part IN {0, 1};
        if part == 0 then
            assert width IN {8,16,32,64};
            return _V[[n]][width-1:0];
        else
            assert width == 64;
            return _V[[n]][(width * 2)-1:width];
        end;
    end;
end;
```

```
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
// part 1 inserts a 64-bit value into the top half of the register.
setter
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        _V[[n]] = ZeroExtend{128}(value);
    else
        assert width == 64;
        _V[[n]][(width * 2)-1:width] = value[width-1:0];
    end;
end;
end;
```

G1.2.3.200 _PC

```
// _PC - the program counter
// =====
```

```
var _PC : bits(64);
```

G1.2.3.201 _R

```
// _R - the general-purpose register file array
// =====
```

```
var _R : array [[31]] of bits(64);
```

G1.2.3.202 _V

```
// _V - the SIMD&FP register file array
// =====
```

```
var _V : array [[32]] of bits(128);
```

G1.2.3.203 SPSR_curr

```
// SPSR_curr - accessor
// =====
```

```
accessor SPSR_curr() <=> value : bits(32)
begin
    getter
        var result : bits(32);
        case PSTATE.M of
            when M32_FIQ    => result = SPSR_fiq() [31:0];
            when M32_IRQ    => result = SPSR_irq() [31:0];
            when M32_Svc    => result = SPSR_svc() [31:0];
            when M32_Abort  => result = SPSR_abt() [31:0];
            when M32_Hyp    => result = SPSR_hyp() [31:0];
            when M32_Undef  => result = SPSR_und() [31:0];
            otherwise      => unreachable;
        end;
        return result;
    end;

    setter
        case PSTATE.M of
            when M32_FIQ    => SPSR_fiq() [31:0] = value[31:0];
            when M32_IRQ    => SPSR_irq() [31:0] = value[31:0];
```

```

        when M32_Svc      => SPSR_svc() [31:0] = value[31:0];
        when M32_Abort    => SPSR_abt() [31:0] = value[31:0];
        when M32_Hyp      => SPSR_hyp() [31:0] = value[31:0];
        when M32_Undef    => SPSR_und() [31:0] = value[31:0];
        otherwise        => unreachable;
    end;
end;
end;

```

G1.2.3.204 AddressAdd

```

// AddressAdd()
// =====
// Add an address with an offset and return the result.

func AddressAdd(base : bits(64), offset : integer, accdesc : AccessDescriptor) => bits(64)
begin
    return AddressAdd(base, offset[63:0], accdesc);
end;

func AddressAdd(base : bits(64), offset : bits(64), accdesc : AccessDescriptor) => bits(64)
begin
    var result : bits(64) = base + offset;
    return result;
end;

```

G1.2.3.205 AddressInNaturallyAlignedBlock

```

// AddressInNaturallyAlignedBlock()
// =====
// Returns TRUE if the given addresses are in the same naturally aligned block, and FALSE otherwise.
// An address is in a naturally aligned block if it meets any of the below conditions:
// * is a power-of-two size.
// * Is no larger than the DC ZVA block size if ESR_ELx.FnP is being set to 0b0, or EDHSR is not
//   implemented or EDHSR.FnP is being set to 0b0 (as appropriate).
// * Is no larger than the smallest implemented translation granule if ESR_ELx.FnP, or EDHSR.FnP
//   (as appropriate) is being set to 0b1.
// * Contains a watchpointed address accessed by the memory access or set of contiguous memory
//   accesses that triggered the watchpoint.

impdef func AddressInNaturallyAlignedBlock(address1 : bits(64), address2 : bits(64)) => boolean
begin
    // Returns TRUE if address in within a naturally aligned block of memory as the watchpointed
    // location, else FALSE.
    return FALSE;
end;

```

G1.2.3.206 AddressIncrement

```

// AddressIncrement()
// =====
// Increment an address and return the result.

func AddressIncrement(base : bits(64), increment : integer, accdesc : AccessDescriptor) => bits(64)
begin
    return AddressIncrement(base, increment[63:0], accdesc);
end;

func AddressIncrement(base : bits(64), increment : bits(64), accdesc : AccessDescriptor) => bits(64)
begin
    var result : bits(64) = base + increment;
    return result;
end;

```

G1.2.3.207 ClearEventRegister

```
// ClearEventRegister()
// =====
// Clear the Event Register of this PE.

func ClearEventRegister()
begin
    EventRegister = '0';
    return;
end;
```

G1.2.3.208 ConditionHolds

```
// ConditionHolds()
// =====
// Return TRUE iff COND currently holds

func ConditionHolds(cond : bits(4)) => boolean
begin
    // Evaluate base condition.
    var result : boolean;
    case cond[3:1] of
        when '000' => result = (PSTATE.Z == '1');           // EQ or NE
        when '001' => result = (PSTATE.C == '1');           // CS or CC
        when '010' => result = (PSTATE.N == '1');           // MI or PL
        when '011' => result = (PSTATE.V == '1');           // VS or VC
        when '100' => result = (PSTATE.C == '1' && PSTATE.Z == '0'); // HI or LS
        when '101' => result = (PSTATE.N == PSTATE.V);      // GE or LT
        when '110' => result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0'); // GT or LE
        when '111' => result = TRUE;                         // AL
    end;

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond[0] == '1' && cond != '1111' then
        result = !result;
    end;

    return result;
end;
```

G1.2.3.209 ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()
// =====

impdef func ConsumptionOfSpeculativeDataBarrier()
begin
    return;
end;
```

G1.2.3.210 CurrentInstrSet

```
// CurrentInstrSet()
// =====

readonly func CurrentInstrSet() => InstrSet
begin
    var result : InstrSet;
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    end;
```

```
    end;  
    return result;  
end;
```

G1.2.3.211 CurrentSecurityState

```
// CurrentSecurityState()  
// =====  
// Returns the effective security state at the exception level based off current settings.  
  
readonly func CurrentSecurityState() => SecurityState  
begin  
    return SS_NonSecure;  
end;
```

G1.2.3.212 DSBAlias

```
// DSBAlias  
// =====  
// Aliases of DSB.  
  
type DSBAlias of enumeration {DSBAlias_SSBB, DSBALias_PSSBB, DSBALias_DSB};
```

G1.2.3.213 EL0

```
// EL0-3  
// =====  
// PSTATE.EL Exception level bits.  
  
constant EL3 : bits(2) = '11';  
constant EL2 : bits(2) = '10';  
constant EL1 : bits(2) = '01';  
constant EL0 : bits(2) = '00';
```

G1.2.3.214 EL2Enabled

```
// EL2Enabled()  
// =====  
// Returns TRUE if EL2 is present and access is Non-secure, FALSE otherwise.  
  
readonly func EL2Enabled() => boolean  
begin  
    return TRUE;  
end;
```

G1.2.3.215 ELFromM32

```
// ELFromM32()  
// =====  
  
func ELFromM32(mode : bits(5)) => (boolean,bits(2))  
begin  
    // Convert an AArch32 mode encoding to an Exception level.  
    // Returns (valid,EL):  
    // 'valid' is TRUE if 'mode[4:0]' encodes a mode that is both valid for this implementation  
    // and the current value of SCR.NS/SCR_EL3.NS.  
    // 'EL' is the Exception level decoded from 'mode'.  
    var el : bits(2);  
    // Check for modes that are not valid for this implementation  
    var valid : boolean = !BadMode(mode);  
  
    case mode of  
        when M32_Monitor =>
```

```

        el = EL3;
    when M32_Hyp =>
        el = EL2;
    when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System =>
        // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
        // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
        // AArch64, then these modes are EL1 modes.
        el = EL1;
    when M32_User =>
        el = EL0;
    otherwise =>
        valid = FALSE;          // Passed an illegal mode value
end;

if !valid then el = ARBITRARY : bits(2); end;
return (valid, el);
end;

```

G1.2.3.216 ELFromSPSR

```

// ELFromSPSR()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr[4:0]' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

func ELFromSPSR{N}(spsr : bits(N)) => (boolean,bits(2))
begin
    var el : bits(2);
    var valid : boolean;
    if spsr[4] == '0' then          // AArch64 state
        el = spsr[3:2];
        if !HaveAArch64() then
            valid = FALSE;          // No AArch64 support
        elsif !HaveEL(el) then
            valid = FALSE;          // Exception level not implemented
        elsif spsr[1] == '1' then
            valid = FALSE;          // M[1] must be 0
        elsif el == EL0 && spsr[0] == '1' then
            valid = FALSE;          // for EL0, M[0] must be 0
        elsif el == EL2 && HaveEL(EL3) && EffectiveSCR_EL3_NS() == '0' then
            valid = FALSE;          // EL2 valid only in Non-secure state
        else
            valid = TRUE;
        end;
    elsif HaveAArch32() then        // AArch32 state
        (valid, el) = ELFromM32(spsr[4:0]);
    else
        valid = FALSE;
    end;

    if !valid then el = ARBITRARY : bits(2); end;
    return (valid,el);
end;

```

G1.2.3.217 ELStateUsingAArch32

```

// ELStateUsingAArch32()
// =====

readonly func ELStateUsingAArch32(el : bits(2), secure : boolean) => boolean
begin

```

```
// See ELStateUsingAArch32K() for description. Must only be called in circumstances where
// result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
let (known, aarch32) : (boolean, boolean) = ELStateUsingAArch32K(el, secure);
assert known;
return aarch32;
end;
```

G1.2.3.218 ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// =====
// Returns (known, aarch32):
//   'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
//           using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
//   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.

readonly func ELStateUsingAArch32K(el : bits(2), secure : boolean) => (boolean, boolean)
begin
    assert HaveEL(el);

    return (TRUE, TRUE);
end;
```

G1.2.3.219 ELUsingAArch32

```
// ELUsingAArch32()
// =====

readonly func ELUsingAArch32(el : bits(2)) => boolean
    recurselimit 1

begin
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
end;
```

G1.2.3.220 ELUsingAArch32K

```
// ELUsingAArch32K()
// =====

readonly func ELUsingAArch32K(el : bits(2)) => (boolean,boolean)
begin
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
end;
```

G1.2.3.221 EffectiveEA

```
// EffectiveEA()
// =====
// Returns the Effective value of SCR_EL3().EA

readonly func EffectiveEA() => bit
begin
    return '0';
end;
```

G1.2.3.222 EffectiveHCR_EL2_NVx

```
// EffectiveHCR_EL2_NVx()
// =====
// Returns the Effective value of HCR_EL2().[NV2,NV1,NV].

func EffectiveHCR_EL2_NVx() => bits(3)
begin
    return '000';
end;
```

G1.2.3.223 EffectiveSCR_EL3_NS

```
// EffectiveSCR_EL3_NS()
// =====
// Returns the Effective value of SCR_EL3().NS.

readonly func EffectiveSCR_EL3_NS() => bit
begin
    return if SecureOnlyImplementation() then '0' else '1';
end;
```

G1.2.3.224 EffectiveSCR_EL3_RW

```
// EffectiveSCR_EL3_RW()
// =====
// Returns the Effective value of SCR_EL3().RW.

readonly func EffectiveSCR_EL3_RW() => bit
begin
    return '0';
end;
```

G1.2.3.225 EffectiveTGE

```
// EffectiveTGE()
// =====
// Returns the Effective value of TGE.

func EffectiveTGE() => bit
begin
    return if EL2Enabled() then HCR().TGE else '0';
end;
```

G1.2.3.226 EndOfInstruction

```
// EndOfInstruction()
// =====
// Terminate processing of the current instruction.

noreturf impdef func EndOfInstruction(EndReason : EndOfInstructionReason)
begin
    throw EndOfInstructionException{reason=EndReason};
end;
```

G1.2.3.227 EndOfInstructionException

```
// EndOfInstructionException
// =====
// End of Instruction exception type.

type EndOfInstructionException of exception {
    reason : EndOfInstructionReason
};
```

G1.2.3.228 EndOfInstructionReason

```
// EndOfInstructionReason
// =====

type EndOfInstructionReason of enumeration {
    EndOfInstructionReason_Retired,
    EndOfInstructionReason_Exception
};
```

G1.2.3.229 EnterLowPowerState

```
// EnterLowPowerState()
// =====
// PE enters a low-power state.

impdef func EnterLowPowerState()
begin
    return;
end;
```

G1.2.3.230 EventRegister

```
// EventRegister
// =====
// Event Register for this PE.

var EventRegister : bits(1);
```

G1.2.3.231 ExceptionalOccurrenceTargetState

```
// ExceptionalOccurrenceTargetState
// =====
// Enumeration to represent the target state of an Exceptional Occurrence.
// The Exceptional Occurrence can be either Exception or Debug State entry.

type ExceptionalOccurrenceTargetState of enumeration {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};
```

G1.2.3.232 ExecuteAsNOP

```
// ExecuteAsNOP()
// =====

func ExecuteAsNOP()
begin
    EndOfInstruction(EndOfInstructionReason_Retired);
end;
```

G1.2.3.233 FIQPending

```
// FIQPending()
// =====
// Returns TRUE if there is any pending physical FIQ.

impdef func FIQPending() => boolean
begin
    return FALSE;
end;
```

G1.2.3.234 GetAccumulatedFPExceptions

```
// GetAccumulatedFPExceptions()
// =====
// Returns FP exceptions accumulated by the PE.

impdef func GetAccumulatedFPExceptions() => bits(8)
begin
    return ARBITRARY : bits(8);
end;
```

G1.2.3.235 GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

func GetPSRFromPSTATE{N}(targetELState : ExceptionalOccurrenceTargetState) => bits(N)
begin
    if UsingAArch32() && (targetELState IN {AArch32_NonDebugState, DebugState}) then
        assert N == 32;
    else
        assert N == 64;
    end;

    var spsr : bits(N) = Zeros{};
    spsr[31:28] = PSTATE.[N,Z,C,V];
    spsr[20] = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        spsr[27] = PSTATE.Q;
        spsr[26:25] = PSTATE.IT[1:0];
        if targetELState IN {AArch64_NonDebugState, DebugState} then
            spsr[21] = PSTATE.SS;
        end;
        spsr[19:16] = PSTATE.GE;
        spsr[15:10] = PSTATE.IT[7:2];
        spsr[9] = PSTATE.E;
        spsr[8:6] = PSTATE.[A,I,F]; // No PSTATE.D in AArch32 state
        spsr[5] = PSTATE.T;
        assert PSTATE.M[4] == PSTATE.nRW; // bit [4] is the discriminator
        spsr[4:0] = PSTATE.M;
    else // AArch64 state
        spsr[21] = PSTATE.SS;
        spsr[9:6] = PSTATE.[D,A,I,F];
        spsr[4] = PSTATE.nRW;
        spsr[3:2] = PSTATE.EL;
        spsr[0] = PSTATE.SP;
    end;
    return spsr;
end;
```

G1.2.3.236 HaveAArch32

```
// HaveAArch32()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.

readonly func HaveAArch32() => boolean
begin
    return IsFeatureImplemented(FEAT_AA32EL0);
end;
```

G1.2.3.237 HaveAArch32EL

```
// HaveAArch32EL()
// =====
// Return TRUE if Exception level 'el' supports AArch32 in this implementation

readonly func HaveAArch32EL(el : bits(2)) => boolean
begin
    case el of
        when EL0 => return IsFeatureImplemented(FEAT_AA32EL0);
        when EL1 => return IsFeatureImplemented(FEAT_AA32EL1);
        when EL2 => return IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 => return FALSE;
    end;
end;
```

G1.2.3.238 HaveAArch64

```
// HaveAArch64()
// =====
// Return TRUE if the highest Exception level is using AArch64 state.

readonly func HaveAArch64() => boolean
begin
    return FALSE;
end;
```

G1.2.3.239 HaveEL

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

readonly func HaveEL(el : bits(2)) => boolean
begin
    case el of
        when EL1, EL0 =>
            return TRUE; // EL1 and EL0 must exist
        when EL2 =>
            return IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 =>
            return FALSE;
        otherwise =>
            unreachable;
    end;
end;
```

G1.2.3.240 HaveSecureState

```
// HaveSecureState()
// =====
// Return TRUE if Secure State is supported.

readonly func HaveSecureState() => boolean
begin
    return SecureOnlyImplementation();
end;
```

G1.2.3.241 HighestEL

```
// HighestEL()
// =====
// Returns the highest implemented Exception level.

readonly func HighestEL() => bits(2)
begin
    if HaveEL(EL3) then
        return EL3;
    elseif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
    end;
end;
```

G1.2.3.242 Hint_Yield

```
// Hint_Yield()
// =====
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.

impdef func Hint_Yield()
begin
    return;
end;
```

G1.2.3.243 IRQPending

```
// IRQPending()
// =====
// Returns TRUE if there is any pending physical IRQ.

impdef func IRQPending() => boolean
begin
    return FALSE;
end;
```

G1.2.3.244 IllegalExceptionReturn

```
// IllegalExceptionReturn()
// =====

func IllegalExceptionReturn{N}(spsr : bits(N)) => boolean
begin
    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    let (valid, target) : (boolean, bits(2)) = ELFromSPSR{N}(spsr);
    if !valid then return TRUE; end;

    // Check for return to higher Exception level.
    if UInt(target) > UInt(PSTATE.EL) then return TRUE; end;

    let spsr_mode_is_aarch32 : boolean = (spsr[4] == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
```



```
// * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
//   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
// * To AArch64 state from AArch32 state (should be caught by above).
let (known, target_el_is_aarch32) : (boolean, boolean) = ELUsingAArch32K(target);
assert known || (target == EL0 && !ELUsingAArch32(EL1));
if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE; end;

// Check for illegal return from AArch32 to AArch64.
if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE; end;
// Check for illegal return to EL1 in Non-secure state when HCR.TGE is set.
if EL2Enabled() && target == EL1 && !IsSecureBelowEL3() && HCR().TGE == '1' then
    return TRUE;
end;

return FALSE;
end;
```

G1.2.3.245 InstrSet

```
// InstrSet
// =====

type InstrSet of enumeration {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

G1.2.3.246 InstructionFetchBarrier

```
// InstructionFetchBarrier()
// =====

impdef func InstructionFetchBarrier()
begin
    return;
end;
```

G1.2.3.247 InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// =====

impdef func InstructionSynchronizationBarrier()
begin
    return;
end;
```

G1.2.3.248 IsASEInstruction

```
// IsASEInstruction()
// =====
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

impdef func IsASEInstruction() => boolean
begin
    return FALSE;
end;
```

G1.2.3.249 IsCurrentSecurityState

```
// IsCurrentSecurityState()
// =====
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

func IsCurrentSecurityState(ss : SecurityState) => boolean
begin
    return CurrentSecurityState() == ss;
end;
```

G1.2.3.250 IsEventRegisterSet

```
// IsEventRegisterSet()
// =====
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

func IsEventRegisterSet() => boolean
begin
    return EventRegister == '1';
end;
```

G1.2.3.251 IsHighestEL

```
// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level implemented

readonly func IsHighestEL(el : bits(2)) => boolean
begin
    return HighestEL() == el;
end;
```

G1.2.3.252 IsSecureBelowEL3

```
// IsSecureBelowEL3()
// =====
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

readonly func IsSecureBelowEL3() => boolean
begin
    return FALSE;
end;
```

G1.2.3.253 IsWFEsleep

```
// IsWFEsleep
// =====
// Set if PE is in low power mode due to WFE(T) instruction

var IsWFEsleep : boolean;
```

G1.2.3.254 IsWFIsleep

```
// IsWFIsleep
// =====
// Set if PE is in low power mode due to WFI(T) instruction

var IsWFIsleep : boolean;
```

G1.2.3.255 Mode

```
// Mode bits
// =====
// AArch32 PSTATE.M mode bits.

constant M32_User : bits(5)    = '10000';
constant M32_FIQ  : bits(5)    = '10001';
constant M32_IRQ  : bits(5)    = '10010';
constant M32_Svc  : bits(5)    = '10011';
constant M32_Monitor : bits(5) = '10110';
constant M32_Abort : bits(5)    = '10111';
constant M32_Hyp  : bits(5)    = '11010';
constant M32_Undef : bits(5)    = '11011';
constant M32_System : bits(5)   = '11111';
```

G1.2.3.256 NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Non-secure for this implementation.

func NonSecureOnlyImplementation() => boolean
begin
    return ImpDefBool("Non-secure only implementation");
end;
```

G1.2.3.257 PLOfEL

```
// PLOfEL()
// =====

func PLOfEL(el : bits(2)) => PrivilegeLevel
begin
    case el of
        when EL3 => return if !HaveAArch64() then PL1 else PL3;
        when EL2 => return PL2;
        when EL1 => return PL1;
        when EL0 => return PL0;
    end;
end;
```

G1.2.3.258 PSTATE

```
// PSTATE
// =====
// PE state bits.
// There is no significance to the field order.

var PSTATE : collection {
    N : bits (1),      // Negative condition flag
    Z : bits (1),      // Zero condition flag
    C : bits (1),      // Carry condition flag
    V : bits (1),      // Overflow condition flag
    D : bits (1),      // Debug mask bit [AArch64 only]
    A : bits (1),      // SError interrupt mask bit
    I : bits (1),      // IRQ mask bit
    F : bits (1),      // FIQ mask bit
    SS : bits (1),     // Software step bit
    IL : bits (1),     // Illegal Execution state bit
    EL : bits (2),     // Exception level
    nRW : bits (1),    // Execution state: 0=AArch64, 1=AArch32
    SP : bits (1),     // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    Q : bits (1),      // Cumulative saturation flag [AArch32 only]
    GE : bits (4),     // Greater than or Equal flags [AArch32 only]
```

```
IT : bits (8),          // If-then bits, RES0 in CPSR          [AArch32 only]
J  : bits (1),          // J bit, RES0                        [AArch32 only, RES0 in SPSR and CPSR]
T  : bits (1),          // T32 bit, RES0 in CPSR              [AArch32 only]
E  : bits (1),          // Endianness bit                     [AArch32 only]
M  : bits (5)           // Mode field                          [AArch32 only]

};
```

G1.2.3.259 PhysicalCountInt

```
// PhysicalCountInt()
// =====
// Returns the integral part of physical count value of the System counter.

func PhysicalCountInt() => bits(64)
begin
    return PhysicalCount[63:0];
end;
```

G1.2.3.260 PrivilegeLevel

```
// PrivilegeLevel
// =====
// Privilege Level abstraction.

type PrivilegeLevel of enumeration {PL3, PL2, PL1, PL0};
```

G1.2.3.261 RestoredITBits

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

func RestoredITBits{N}(spsr : bits(N)) => bits(8)
begin
    let it : bits(8) = spsr[15:10,26:25];

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;
    end;
end;

// The IT bits are forced to zero when they are set to a reserved value.
if !IsZero(it[7:4]) && IsZero(it[3:0]) then
    return '00000000';
end;

// The IT bits are forced to zero when returning to A32 state, or when returning to an EL
// with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
let itd : bit = if PSTATE.EL == EL2 then HSCTLR().ITD else SCTLR().ITD;
if (spsr[5] == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it[2:0])) then
    return '00000000';
else
    return it;
end;
end;
```

G1.2.3.262 SecureOnlyImplementation

```
// SecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Secure for this implementation.

readonly func SecureOnlyImplementation() => boolean
begin
    return ImpDefBool("Secure-only implementation");
end;
```

G1.2.3.263 SecurityState

```
// SecurityState
// =====
// The Security state of an execution context

type SecurityState of enumeration {
    SS_NonSecure,
    SS_Secure
};
```

G1.2.3.264 SecurityStateAtEL

```
// SecurityStateAtEL()
// =====
// Returns the effective security state at the exception level based off current settings.

readonly func SecurityStateAtEL(EL : bits(2)) => SecurityState
begin
    return SS_NonSecure;
end;
```

G1.2.3.265 SendEvent

```
// SendEvent()
// =====
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.

impdef func SendEvent()
begin
    SendEventLocal();
end;
```

G1.2.3.266 SendEventLocal

```
// SendEventLocal()
// =====
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

func SendEventLocal()
begin
    EventRegister = '1';
    return;
end;
```

G1.2.3.267 SetAccumulatedFPExceptions

```
// SetAccumulatedFPExceptions()
// =====
// Stores FP Exceptions accumulated by the PE.

impdef func SetAccumulatedFPExceptions(accumulated_exceptions : bits(8))
begin
    return;
end;
```

G1.2.3.268 SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// =====

func SetPSTATEFromPSR{N}(spsr : bits(N))
begin
    let illegal_psr_state : boolean = IllegalExceptionReturn{N}(spsr);
    SetPSTATEFromPSR{N}(spsr, illegal_psr_state);
end;

// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

func SetPSTATEFromPSR{N}(spsr_in : bits(N), illegal_psr_state : boolean)
begin
    var spsr : bits(N) = spsr_in;
    let from_aarch64 : boolean = FALSE;
    PSTATE.SS = DebugExceptionReturnSS{N}(spsr);
    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr[20];
        if spsr[4] == '1' then // AArch32 state
            AArch32_WriteMode(spsr[4:0]); // Sets PSTATE.EL correctly
        else // AArch64 state
            PSTATE.nRW = '0';
            (-, PSTATE.EL) = ELFromSPSR{N}(spsr);
            PSTATE.SP = spsr[0];
        end;
    end;

    // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
    // copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr[5] = '0'; end;
    end;

    // State that is reinstated regardless of illegal exception return
    PSTATE.[N,Z,C,V] = spsr[31:28];
    if PSTATE.nRW == '1' then // AArch32 state
        PSTATE.Q = spsr[27];
        PSTATE.IT = RestoredITBits{N}(spsr);
        ShouldAdvanceIT = FALSE;
        PSTATE.GE = spsr[19:16];
        PSTATE.E = spsr[9];
        PSTATE.[A,I,F] = spsr[8:6]; // No PSTATE.D in AArch32 state
        PSTATE.T = spsr[5]; // PSTATE.J is RES0
    else // AArch64 state
        PSTATE.[D,A,I,F] = spsr[9:6]; // No PSTATE.[Q,IT,GE,E,T] in AArch64 state
    end;
    return;
```

```
end;
```

G1.2.3.269 ShouldAdvanceHS

```
// ShouldAdvanceHS
// =====
// Cleared if we should not advance the EDES.RS after the current instruction.

var ShouldAdvanceHS : boolean;
```

G1.2.3.270 ShouldAdvanceIT

```
// ShouldAdvanceIT
// =====
// Cleared if we should not advance the PSTATE.IT after the current instruction.

var ShouldAdvanceIT : boolean;
```

G1.2.3.271 ShouldAdvanceSS

```
// ShouldAdvanceSS
// =====
// Cleared if PSTATE.SS is written by the current instruction.

var ShouldAdvanceSS : boolean;
```

G1.2.3.272 SynchronizeContext

```
// SynchronizeContext()
// =====
// Context Synchronization event, includes Instruction Fetch Barrier effect

impdef func SynchronizeContext()
begin
    InstructionFetchBarrier();
end;
```

G1.2.3.273 TakeUnmaskedPhysicalSErrorInterrupts

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt.

impdef func TakeUnmaskedPhysicalSErrorInterrupts(iesb_req : boolean)
begin
    return;
end;
```

G1.2.3.274 TakeUnmaskedSErrorInterrupts

```
// TakeUnmaskedSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.

impdef func TakeUnmaskedSErrorInterrupts()
begin
    return;
end;
```

G1.2.3.275 ThisInstr

```
// ThisInstr()
// =====
// Return the bitstring encoding of the currently executing instruction.

impdef func ThisInstr() => bits(32)
begin
    Unimplemented();
end;
```

G1.2.3.276 ThisInstrLength

```
// ThisInstrLength()
// =====
// Return the length of the current instruction.

impdef func ThisInstrLength() => integer
begin
    Unimplemented();
end;
```

G1.2.3.277 UsingAArch32

```
// UsingAArch32()
// =====
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64_

readonly func UsingAArch32() => boolean
begin
    let aarch32 : boolean = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32; end;
    if !HaveAArch64() then assert aarch32; end;
    return aarch32;
end;
```

G1.2.3.278 ValidSecurityStateAtEL

```
// ValidSecurityStateAtEL()
// =====
// Returns TRUE if the current settings and architecture choices for this
// implementation permit a valid Security state at the indicated EL.

func ValidSecurityStateAtEL(el : bits(2)) => boolean
begin
    if !HaveEL(el) then
        return FALSE;
    end;

    if el == EL2 then
        return EL2Enabled();
    end;

    return TRUE;
end;
```

G1.2.3.279 VirtualFIQPending

```
// VirtualFIQPending()
// =====
// Returns TRUE if there is any pending virtual FIQ.

impdef func VirtualFIQPending() => boolean
begin
    return FALSE;
end;
```

G1.2.3.280 VirtualIRQPending

```
// VirtualIRQPending()
// =====
// Returns TRUE if there is any pending virtual IRQ.

impdef func VirtualIRQPending() => boolean
begin
    return FALSE;
end;
```

G1.2.3.281 WFxType

```
// WFxType
// =====
// WFx instruction types.

type WFxType of enumeration {WFxType_WFE, WFxType_WFI};
```

G1.2.3.282 WaitForEvent

```
// WaitForEvent()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

func WaitForEvent()
begin
    if Halted() then return; end;
    if !IsEventRegisterSet() then
        EnterLowPowerState();
    end;
    return;
end;
```

G1.2.3.283 WaitForInterrupt

```
// WaitForInterrupt()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.

func WaitForInterrupt()
begin
    if Halted() then return; end;
    EnterLowPowerState();
    return;
end;
```

G1.2.3.284 WatchpointRelatedSyndrome

```
// WatchpointRelatedSyndrome()
// =====
// Update common Watchpoint related fields.

func WatchpointRelatedSyndrome(fault : FaultRecord) => bits(24)
begin
    var syndrome : bits(24) = Zeros{};

    return syndrome;
end;
```

G1.2.3.285 tlbi

```
constant VMID_NONE : bits(NUM_VMIDBITS) = Zeros{};
constant ASID_NONE : bits(NUM_ASIDBITS) = Zeros{};
```

G1.2.3.286 Broadcast

```
// Broadcast
// =====

type Broadcast of enumeration {
    Broadcast_ISH,
    Broadcast_ForcedISH,
    Broadcast_OSH,
    Broadcast_NSH
};
```

G1.2.3.287 BroadcastTLBI

```
// BroadcastTLBI()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated broadcast
// domain.

impdef func BroadcastTLBI(broadcast : Broadcast, r : TLBIRecord)
begin
    return;
end;
```

G1.2.3.288 TLBI

```
// TLBI()
// =====
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

impdef func TLBI(r : TLBIRecord)
begin
    return;
end;
```

G1.2.3.289 TLBILevel

```
// TLBILevel
// =====

type TLBILevel of enumeration {
    TLBILevel_Any,        // this applies to TLB entries at all levels
    TLBILevel_Last       // this applies to TLB entries at last level only
};
```

G1.2.3.290 TLBIMemAttr

```
// TLBIMemAttr
// =====
// Defines the attributes of the memory operations that must be completed in
// order to deem the TLBI operation as completed.

type TLBIMemAttr of enumeration {
    TLBI_AllAttr,          // All TLB entries within the scope of the invalidation
    TLBI_ExcludeXS        // Only TLB entries with XS=0 within the scope of the invalidation
};
```

G1.2.3.291 TLBIOp

```
// TLBIOp
// =====

type TLBIOp of enumeration {
    TLBIOp_DALL,          // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,          // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
};
```

G1.2.3.292 TLBIRecord

```
// TLBIRecord
// =====
// Details related to a TLBI operation.

type TLBIRecord of record {
    op : TLBIOp,
    from_aarch64 : boolean, // originated as an AArch64 operation
    security : SecurityState,
    regime : Regime,
    use_vmid : boolean,
    vmid : bits(NUM_VMIDBITS),
    asid : bits(NUM_ASIDBITS),
    level : TLBILevel,
    attr : TLBIMemAttr,
    ipaspace : PAspace,     // For operations that take IPA as input address
    address : bits(64),     // input address, for range operations, start address
    end_address : bits(64), // for range operations, end; address
    tg : bits(2)            // for range operations, translation granule
};
```

G1.2.3.293 VMID

```
// VMID
// ====
// Effective VMID.

func VMID() => bits(NUM_VMIDBITS)
```

```
begin
    return ZeroExtend{NUM_VMIDBITS}(VSCTLR().VMID);
end;
```

G1.2.3.294 Maximum

```
// Maximum walk levels
// =====
constant MAX_WALK_LEVELS : integer{} = 4;
```

G1.2.3.295 Unbounded

```
// Unbounded Variables
// =====
// Sets the upper limit for a specific loop or function.

constant Unbounded : integer{} = 2^128;
config Unbounded_DiscardTransactionalWrites : integer{0..Unbounded} = Unbounded;
config Unbounded_DescriptorUpdate : integer{0..Unbounded} = Unbounded;
config Unbounded_semihost : integer{0..Unbounded} = Unbounded;
config Unbounded_ParseTrace : integer{0..Unbounded} = Unbounded;
config Unbounded_ParseExtensionPacket : integer{0..Unbounded} = Unbounded;
config Unbounded_ULEB128 : integer{0..Unbounded} = Unbounded;
config Unbounded_ULEB128n : integer{0..Unbounded} = Unbounded;
config Unbounded_FloorPow2 : integer{0..Unbounded} = Unbounded;
config Unbounded_NormalizeReal : integer{0..Unbounded} = Unbounded;
```

G1.2.3.296 ConstrainUnpredictable

```
// ConstrainUnpredictable()
// =====
// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

readonly impdef func ConstrainUnpredictable(which : Unpredictable) => Constraint
begin
    case which of
        when Unpredictable_VMSR =>
            return Constraint_UNDEF;
        when Unpredictable_WBOVERLAPLD =>
            return Constraint_WBSUPPRESS; // return loaded value
        when Unpredictable_WBOVERLAPST =>
            return Constraint_NONE; // store pre-writeback value
        when Unpredictable_LDPOVERLAP =>
            return Constraint_UNDEF; // instruction is UNDEFINED
        when Unpredictable_BASEOVERLAP =>
            return Constraint_UNKNOWN; // use UNKNOWN address
        when Unpredictable_DATAOVERLAP =>
            return Constraint_UNKNOWN; // store UNKNOWN value
        when Unpredictable_DEVPAGE2 =>
            return Constraint_FAULT; // take an alignment fault
        when Unpredictable_INSTRDEVICE =>
            return Constraint_NONE; // Do not take a fault
        when Unpredictable_RESCPACR =>
            return Constraint_TRUE; // Map to UNKNOWN value
        when Unpredictable_RESMAIR =>
            return Constraint_UNKNOWN; // Map to UNKNOWN value
        when Unpredictable_S2RESMEMATTR =>
            return Constraint_NC; // Map to Noncacheable value
        when Unpredictable_RESTEXCB =>
            return Constraint_UNKNOWN; // Map to UNKNOWN value
        when Unpredictable_RESDACR =>
```

```

        return Constraint_UNKNOWN; // Map to UNKNOWN value
when Unpredictable_RESPRRR =>
    return Constraint_UNKNOWN; // Map to UNKNOWN value
when Unpredictable_RESVTCRS =>
    return Constraint_UNKNOWN; // Map to UNKNOWN value
when Unpredictable_REStnSZ =>
    return Constraint_FORCE; // Map to the limit value
when Unpredictable_ESRCONDPASS =>
    return Constraint_FALSE; // Report as "AL"
when Unpredictable_ILZEROIT =>
    return Constraint_FALSE; // Do not zero PSTATE.IT
when Unpredictable_ILZEROT =>
    return Constraint_FALSE; // Do not zero PSTATE.T
when Unpredictable_BPVECTORCATCHPRI =>
    return Constraint_TRUE; // Debug Vector Catch: match on 2nd halfword
when Unpredictable_VCMATCHHALF =>
    return Constraint_FALSE; // No match
when Unpredictable_VCMATCHDAPA =>
    return Constraint_FALSE; // No match on Data Abort or Prefetch abort
when Unpredictable_WPMASKANDEAS =>
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_WPBASCONTIGUOUS =>
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_RESWPMASK =>
    return Constraint_DISABLED; // Watchpoint disabled
when Unpredictable_WPMASKEDBITS =>
    return Constraint_FALSE; // Watchpoint disabled
when Unpredictable_RESBPWPCTRL =>
    return Constraint_DISABLED; // Breakpoint/watchpoint disabled
when Unpredictable_BPNOTIMPL =>
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_RESBPTYPE =>
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPNOTCTXCMP =>
    return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPMATCHHALF =>
    return Constraint_FALSE; // No match
when Unpredictable_BPMISMATCHHALF =>
    return Constraint_FALSE; // No match
when Unpredictable_BPLINKINGDISABLED =>
    return Constraint_FALSE; // Breakpoint does not match
when Unpredictable_RESTARTALIGNPC =>
    return Constraint_TRUE; // Force alignment
when Unpredictable_RESTARTZEROUPPERPC =>
    return Constraint_TRUE; // Force zero extension
when Unpredictable_ZEROUPPER =>
    return Constraint_TRUE; // zero top halves of X registers
when Unpredictable_ERETZEROUPPERPC =>
    return Constraint_TRUE; // zero top half of PC
when Unpredictable_A32FORCEALIGNPC =>
    return Constraint_FALSE; // Do not force alignment
when Unpredictable_SMD =>
    return Constraint_UNDEF; // disabled SMC is Unallocated
when Unpredictable_Shareability =>
    return Constraint_OSH; // Map reserved encoding of shareability to Outer Shareable
when Unpredictable_COUNT_CHAIN => // Chained PMU counters idx, idx+1 are not in same range
    return Constraint_FALSE;
when Unpredictable_CLEARERRITEZERO => // Clearing sticky errors when instruction in flight
    return Constraint_FALSE;
when Unpredictable_ALVEXCEPTIONRETURN =>
    return Constraint_UNDEF;
when Unpredictable_DBGxVR_RESS =>
    return Constraint_FALSE;
when Unpredictable_PMSCR_PCT =>
    return Constraint_PMSCR_PCT_VIRT;
when Unpredictable_IGNORETRAPINDEBUG =>

```

```

        return Constraint_FALSE;    // Trap to register access in debug state is enabled
    when Unpredictable_PMUEVENTCOUNTER =>
        return Constraint_UNDEF;    // Accesses to the register are UNDEFINED
    when Unpredictable_RES_HPMN =>
        return Constraint_UNKNOWN;  // The counter is reserved for EL2 use
    when Unpredictable_Atomic_MMU_IMPDEF_FAULT =>
        return Constraint_FALSE;
    when Unpredictable_BankedRegister =>
        return Constraint_NOP;
    when Unpredictable_UnimplementedRegister =>
        return Constraint_NOP;
    when Unpredictable_BPNOTLINKING =>
        return Constraint_FALSE;
    otherwise =>
        unreachable;
end;
end;

```

G1.2.3.297 ConstrainUnpredictableBits

```

// ConstrainUnpredictableBits()
// =====

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

readonly impdef func ConstrainUnpredictableBits(width)(which : Unpredictable
                                                                ) => (Constraint, bits(width))
begin
    let c : Constraint = ConstrainUnpredictable(which);

    case c of
        when Constraint_UNKNOWN =>
            return (c, Zeros{width});    // See notes; this is an example implementation only
        when Constraint_PMSCR_PCT_VIRT =>
            return (c, Zeros{width});
        when Constraint_NC where which == Unpredictable_S2RESMEMATTR =>
            assert width == 4;
            return (c, ZeroExtend{width}('0101'));
        when Constraint_WT where which == Unpredictable_S2RESMEMATTR =>
            assert width == 4;
            return (c, ZeroExtend{width}('1010'));
        when Constraint_WB where which == Unpredictable_S2RESMEMATTR =>
            assert width == 4;
            return (c, ZeroExtend{width}('1111'));
        otherwise =>
            return (c, ARBITRARY : bits(width));    // bits result not used
    end;
end;

```

G1.2.3.298 ConstrainUnpredictableBool

```

// ConstrainUnpredictableBool()
// =====

// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.

impdef func ConstrainUnpredictableBool(which : Unpredictable) => boolean
begin
    let c : Constraint = ConstrainUnpredictable(which);
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
end;

```

G1.2.3.299 ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// =====
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.

impdef func ConstrainUnpredictableInteger(low : integer,
                                         high : integer,
                                         which : Unpredictable) => (Constraint, integer)

begin
    let c : Constraint = ConstrainUnpredictable(which);
    if c == Constraint_UNKNOWN then
        return (c, low); // See notes; this is an example implementation only
    else
        return (c, ARBITRARY : integer); // integer result not used
    end;
end;
```

G1.2.3.300 ConstrainUnpredictableProcedure

```
// ConstrainUnpredictableProcedure()
// =====
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable situation,
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED manner.
// The details of possible outcomes may not be present in the code and must be interpreted
// for each use with respect to the CONSTRAINED UNPREDICTABLE specifications
// for the specific area.

impdef func ConstrainUnpredictableProcedure(which : Unpredictable)

begin
    case which of
        when Unpredictable_PMUEVENTCOUNTER =>
            Undefined();
        when Unpredictable_MRC_APSR_TARGET =>
            PSTATE.[N,Z,C,V] = DBGDSCRint() [31:28];
        when Unpredictable_UnimplementedMPURregion_RW_UNKNOWN =>
            return;
        otherwise =>
            ImpDef("IMPLEMENTATION_DEFINED");
    end;
    return;
end;
```

G1.2.3.301 Constraint

```
// Constraint
// =====
// List of Constrained Unpredictable behaviors.

type Constraint of enumeration {
    // General
    Constraint_NONE, // Instruction executes with
                    // no change or side-effect
                    // to its described behavior
    Constraint_UNKNOWN, // Destination register
                    // has UNKNOWN value
    Constraint_UNDEF, // Instruction is UNDEFINED
    Constraint_UNDEFEL0, // Instruction is UNDEFINED at EL0 only
    Constraint_NOP, // Instruction executes as NOP
    Constraint_TRUE,
```

```

Constraint_FALSE,
Constraint_DISABLED,
Constraint_UNCOND,           // Instruction executes unconditionally
Constraint_COND,            // Instruction executes conditionally
Constraint_ADDITIONAL_DECODE, // Instruction executes
                               // with additional decode

// Load-store
Constraint_WBSUPPRESS,
Constraint_FAULT,
Constraint_MPU_FAULT,        // Raise MPU fault
Constraint_MPU_ATTR_UNKNOWN, // MPU Attribute is UNKNOWN
Constraint_OSH,              // Constrain to Outer Shareable
Constraint_ISH,              // Constrain to Inner Shareable
Constraint_NSH,              // Constrain to Nonshareable

Constraint_NC,               // Constrain to Noncacheable
Constraint_WT,               // Constrain to Writethrough
Constraint_WB,               // Constrain to Writeback

// IPA too large
Constraint_FORCE, Constraint_FORCENOSLICK,
// PMSCR_PCT reserved values select Virtual timestamp
Constraint_PMSCR_PCT_VIRT,
Constraint_ANYREG,           // Any allocated register is chosen
};

```

G1.2.3.302 Unpredictable

```

// Unpredictable
// =====
// List of Constrained Unpredictable situations.

type Unpredictable of enumeration {
    // VMSR on MVFR
    Unpredictable_VMSR,
    // Writeback/transfer register overlap (load)
    Unpredictable_WBOVERLAPLD,
    // Writeback/transfer register overlap (store)
    Unpredictable_WBOVERLAPST,
    // Load Pair transfer register overlap
    Unpredictable_LDPOVERLAP,
    // Store-exclusive base/status register overlap
    Unpredictable_BASEOVERLAP,
    // Store-exclusive data/status register overlap
    Unpredictable_DATAOVERLAP,
    // Load-store alignment checks
    Unpredictable_DEVPAGE2,
    // Instruction fetch from Device memory
    Unpredictable_INSTRDEVICE,
    // Reserved CPACR value
    Unpredictable_RESCPACR,
    // Reserved MAIR value
    Unpredictable_RESMAIR,
    // Reserved Stage 2 MemAttr value
    Unpredictable_S2RESMEMATTR,
    // Reserved TEX::C::B value
    Unpredictable_RESTEXCB,
    // Reserved PRRR value
    Unpredictable_RESPRRR,
    // Reserved DACR field
    Unpredictable_RESDACR,
    // Reserved VTCR.S value
    Unpredictable_RESVTCRS,
    // Reserved or out-of-range TCR.TnSZ value
    Unpredictable_RESTnSZ,
    // Syndrome for a known-passing conditional A32 instruction
}

```



```

Unpredictable_ESRCONDPASS,
// Illegal State exception: zero PSTATE.IT
Unpredictable_ILZEROIT,
// Illegal State exception: zero PSTATE.T
Unpredictable_ILZEROT,
// Debug: prioritization of Vector Catch
Unpredictable_BPVECTORCATCHPRI,
// Debug Vector Catch: match on 2nd halfword
Unpredictable_VCMATCHHALF,
// Debug Vector Catch: match on Data Abort
// or Prefetch abort
Unpredictable_VCMATCHDAPA,
// Debug watchpoints: nonzero MASK and non-ones BAS
Unpredictable_WPMASKANDBAS,
// Debug watchpoints: non-contiguous BAS
Unpredictable_WPBASCONTIGUOUS,
// Debug watchpoints: reserved MASK
Unpredictable_RESWPMASK,
// Debug watchpoints: nonzero MASKed bits of address
Unpredictable_WPMASKEDBITS,
// Debug breakpoints and watchpoints: reserved control bits
Unpredictable_RESBPWPCTRL,
// Debug breakpoints: not implemented
Unpredictable_BPNOTIMPL,
// Debug breakpoints: reserved type
Unpredictable_RESBPTYPE,
// Debug breakpoints: not-context-aware breakpoint
Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,
// Debug breakpoints: a breakpoint is linked to that is not
// programmed with linking enabled
Unpredictable_BPLINKINGDISABLED,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to
// AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking
// branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// Stage-2 EL1&0 mismatch MPU configuration.
Unpredictable_S2EL1MPUCONFIG,
// Reserved shareability encoding
Unpredictable_Shareability,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRRITZZERO,
// ALUEXCEPTIONRETURN when in user/system mode in
// A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGxVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
// Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
// Reserved PMSCR.PCT behavior
Unpredictable_PMSCR_PCT,

```

```

// MDCR_EL2.HPMN or HDCR.HPMN is larger than PMCR.N or
// FEAT_HPMN0 is not implemented and HPMN is 0.
Unpredictable_RES_HPMN,
// Chained PMU counters idx and idx+1 are not in same range
Unpredictable_COUNT_CHAIN,
// Any access to MPU region register above the number of
// implemented regions specified by
// MPUIR_ELx.REGION/H(MPUIR).REGION is constrained unpredictable
Unpredictable_UnimplementedMPURegion_RW_UNKNOWN,
Unpredictable_Atomic_MMU_IMPDEF_FAULT,
// Accessing DBGDSRint via MRC in debug state
Unpredictable_MRC_APSR_TARGET,
// Accessing Banked register not accessible from the PE mode
Unpredictable_BankedRegister,
// Accessing unimplemented Banked register
Unpredictable_UnimplementedRegister,
// Linked target does not support linking
Unpredictable_BPNOTLINKING
};

```

G1.2.3.303 AdvSIMDEExpandImm

```

// AdvSIMDEExpandImm()
// =====

func AdvSIMDEExpandImm(op : bit, cmode : bits(4), imm8 : bits(8)) => bits(64)
begin
    var imm64 : bits(64);
    case cmode[3:1] of
        when '000' =>
            imm64 = Replicate{64}(Zeros{24}::imm8);
        when '001' =>
            imm64 = Replicate{64}(Zeros{16}::imm8::Zeros{8});
        when '010' =>
            imm64 = Replicate{64}(Zeros{8}::imm8::Zeros{16});
        when '011' =>
            imm64 = Replicate{64}(imm8::Zeros{24});
        when '100' =>
            imm64 = Replicate{64}(Zeros{8}::imm8);
        when '101' =>
            imm64 = Replicate{64}(imm8::Zeros{8});
        when '110' =>
            if cmode[0] == '0' then
                imm64 = Replicate{64}(Zeros{16}::imm8::Ones{8});
            else
                imm64 = Replicate{64}(Zeros{8}::imm8::Ones{16});
            end;
        when '111' =>
            if cmode[0] == '0' && op == '0' then
                imm64 = Replicate{64}(imm8);
            end;
            if cmode[0] == '0' && op == '1' then
                let imm8a : bits(8) = Replicate{}(imm8[7]);
                let imm8b : bits(8) = Replicate{}(imm8[6]);
                let imm8c : bits(8) = Replicate{}(imm8[5]);
                let imm8d : bits(8) = Replicate{}(imm8[4]);
                let imm8e : bits(8) = Replicate{}(imm8[3]);
                let imm8f : bits(8) = Replicate{}(imm8[2]);
                let imm8g : bits(8) = Replicate{}(imm8[1]);
                let imm8h : bits(8) = Replicate{}(imm8[0]);
                imm64 = imm8a::imm8b::imm8c::imm8d::imm8e::imm8f::imm8g::imm8h;
            end;
            if cmode[0] == '1' && op == '0' then
                let imm32 : bits(32) = (imm8[7]::NOT(imm8[6])::Replicate{5}(imm8[6])::
                    imm8[5:0]::Zeros{19});
                imm64 = Replicate{64}(imm32);
            end;
    end
end

```

```

end;
if cmode[0] == '1' && op == '1' then
    if UsingAArch32() then ReservedEncoding(); end;
    imm64 = imm8[7]::NOT(imm8[6])::Replicate{8}(imm8[6])::imm8[5:0]::Zeros{48};
end;
end;

return imm64;
end;

```

G1.2.3.304 PolynomialMult

```

// PolynomialMult()
// =====

func PolynomialMult{M, N}(op1 : bits(M), op2 : bits(N)) => bits(M+N)
begin
    var result : bits(N + M) = Zeros{M+N};
    let extended_op2 : bits(N + M) = ZeroExtend{M+N}(op2);
    for i=0 to M-1 do
        if op1[i] == '1' then
            result = result XOR LSL(extended_op2, i);
        end;
    end;
    return result;
end;

```

G1.2.3.305 SatQ

```

// SatQ()
// =====

func SatQ{N}(i : integer, unsigned : boolean) => (bits(N), boolean)
begin
    let (result, sat) : (bits(N), boolean) = (if unsigned then UnsignedSatQ{N}(i)
                                                else SignedSatQ{N}(i));
    return (result, sat);
end;

```

G1.2.3.306 ShiftSat

```

// ShiftSat()
// =====

func ShiftSat(shift : integer, esize : integer) => integer
begin
    if shift > esize+1 then return esize+1;
    elsif shift < -(esize+1) then return -(esize+1); end;
    return shift;
end;

```

G1.2.3.307 SignedSat

```

// SignedSat()
// =====

func SignedSat{N}(i : integer) => bits(N)
begin
    let (result, -) = SignedSatQ{N}(i);
    return result;
end;

```

G1.2.3.308 SignedSatQ

```
// SignedSatQ()
// =====

func SignedSatQ{N}(i : integer) => (bits(N), boolean)
begin
    var result : integer;
    var saturated : boolean;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1; saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1)); saturated = TRUE;
    else
        result = i; saturated = FALSE;
    end;
    return (result[N-1:0], saturated);
end;
```

G1.2.3.309 UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =====

func UnsignedRSqrtEstimate{N}(operand : bits(N)) => bits(N)
begin
    assert N == 32;
    var result : bits(N);
    if operand[N-1:N-2] == '00' then // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones{N};
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        let estimate : integer = RecipSqrtEstimate(UInt(operand[31:23]));
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate[8:0] :: Zeros{N-9};
    end;

    return result;
end;
```

G1.2.3.310 UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =====

func UnsignedRecipEstimate{N}(operand : bits(N)) => bits(N)
begin
    assert N == 32;
    var result : bits(N);
    if operand[N-1] == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones{N};
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        let estimate : integer = RecipEstimate(UInt(operand[31:23]));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate[8:0] :: Zeros{N-9};
    end;

    return result;
end;
```

G1.2.3.311 UnsignedSat

```
// UnsignedSat()
// =====

func UnsignedSat{N}(i : integer) => bits(N)
begin
    let (result, -) = UnsignedSatQ{N}(i);
    return result;
end;
```

G1.2.3.312 UnsignedSatQ

```
// UnsignedSatQ()
// =====

func UnsignedSatQ{N}(i : integer) => (bits(N), boolean)
begin
    var result : integer;
    var saturated : boolean;
    if i > 2^N - 1 then
        result = 2^N - 1; saturated = TRUE;
    elsif i < 0 then
        result = 0; saturated = TRUE;
    else
        result = i; saturated = FALSE;
    end;
    return (result[N-1:0], saturated);
end;
```

G1.2.4 shared/trace

This section includes the following pseudocode functions:

- [shared/translation](#)

G1.2.5 shared/translation

This section includes the following pseudocode functions:

- [at](#)
- [EncodePARAttrs](#)
- [PAREncodeShareability](#)
- [ReportedPARAttrs](#)
- [ReportedPARShareability](#)
- [DecodeDevice](#)
- [DecodeLDFAAttr](#)
- [DecodeShareability](#)
- [EffectiveShareability](#)
- [IsWBShareable](#)
- [NormalNCMemAttr](#)
- [S1ConstrainUnpredictableRESMAIR](#)
- [S1DecodeMemAttrs](#)
- [S2CombineS1AttrHints](#)
- [S2CombineS1Device](#)
- [S2CombineS1MemAttrs](#)

- [S2CombineS1Shareability](#)
- [S2DecodeCacheability](#)
- [S2DecodeMemAttrs](#)
- [S2ResMemAttr](#)
- [AlignmentFault](#)
- [ExclusiveFault](#)
- [NoFault](#)
- [S1TranslationRegime](#)
- [AddressDescriptor](#)
- [CreateAddressDescriptor](#)
- [CreateFaultyAddressDescriptor](#)
- [HasUnprivileged](#)
- [NUM_PABITS](#)
- [Regime](#)
- [TLBContext](#)
- [TLBRecord](#)
- [TranslationRegime](#)

G1.2.5.1 at

```
type TranslationStage of enumeration {
    TranslationStage_1,
    TranslationStage_12
};

type ATAccess of enumeration {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```

G1.2.5.2 EncodePARAttrs

```
// EncodePARAttrs()
// =====
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

func EncodePARAttrs(memattrs : MemoryAttributes) => bits(8)
begin
    var result : bits(8);

    if memattrs.memtype == MemType\_Device then
        result[7:4] = '0000';
        case memattrs.device of
            when DeviceType\_nGnRnE => result[3:0] = '0000';
            when DeviceType\_nGnRE  => result[3:0] = '0100';
            when DeviceType\_nGRE   => result[3:0] = '1000';
            when DeviceType\_GRE    => result[3:0] = '1100';
            otherwise              => unreachable;
        end;
    else
        if memattrs.outer.attrs == MemAttr\_WT then
            result[7:6] = if memattrs.outer.transient then '00' else '10';
            result[5:4] = memattrs.outer.hints;
        elsif memattrs.outer.attrs == MemAttr\_WB then
            result[7:6] = if memattrs.outer.transient then '01' else '11';
            result[5:4] = memattrs.outer.hints;
```

```

        else // MemAttr_NC
            result[7:4] = '0100';
        end;

        if memattrs.inner.attrs == MemAttr_WT then
            result[3:2] = if memattrs.inner.transient then '00' else '10';
            result[1:0] = memattrs.inner.hints;
        elsif memattrs.inner.attrs == MemAttr_WB then
            result[3:2] = if memattrs.inner.transient then '01' else '11';
            result[1:0] = memattrs.inner.hints;
        else // MemAttr_NC
            result[3:0] = '0100';
        end;
    end;

    return result;
end;

```

G1.2.5.3 PAREncodeShareability

```

// PAREncodeShareability()
// =====
// Derive 64-bit PAR SH field.

func PAREncodeShareability(memattrs : MemoryAttributes) => bits(2)
begin
    if (memattrs.memtype == MemType_Device ||
        (memattrs.inner.attrs == MemAttr_NC &&
         memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';
    end;

    case memattrs.shareability of
        when Shareability_NSH => return '00';
        when Shareability_ISH => return '11';
        when Shareability_OSH => return '10';
    end;
end;

```

G1.2.5.4 ReportedPARAttrs

```

// ReportedPARAttrs()
// =====
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

impdef func ReportedPARAttrs(parattrs : bits(8)) => bits(8)
begin
    return parattrs;
end;

```

G1.2.5.5 ReportedPARShareability

```

// ReportedPARShareability()
// =====
// The value returned in SH field can be the resulting attribute, as determined by any
// permitted implementation choices and any applicable configuration bits, instead of
// the value that appears in the translation table descriptor.

impdef func ReportedPARShareability(sh : bits(2)) => bits(2)
begin
    return sh;
end;

```

G1.2.5.6 DecodeDevice

```
// DecodeDevice()
// =====
// Decode output Device type

func DecodeDevice(device : bits(2)) => DeviceType
begin
    case device of
        when '00' => return DeviceType_nGnRnE;
        when '01' => return DeviceType_nGnRE;
        when '10' => return DeviceType_nGRE;
        when '11' => return DeviceType_GRE;
    end;
end;
```

G1.2.5.7 DecodeLDFAttr

```
// DecodeLDFAttr()
// =====
// Decode memory attributes using LDF (Long Descriptor Format) mapping

func DecodeLDFAttr(attr : bits(4)) => MemAttrHints
begin
    var ldffattr : MemAttrHints;

    if attr == 'x0xx' then ldffattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldffattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr == 'x1xx' then ldffattr.attrs = MemAttr_WB; // Write-back
    else
        unreachable;
    end;

    // Allocation hints are applicable only to cacheable memory.
    if ldffattr.attrs != MemAttr_NC then
        case attr[1:0] of
            when '00' => ldffattr.hints = MemHint_No; // No allocation hints
            when '01' => ldffattr.hints = MemHint_WA; // Write-allocate
            when '10' => ldffattr.hints = MemHint_RA; // Read-allocate
            when '11' => ldffattr.hints = MemHint_RWA; // Read/Write allocate
        end;
    end;

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldffattr.attrs != MemAttr_NC && ldffattr.hints != MemHint_No then
        ldffattr.transient = attr[3] == '0';
    end;

    return ldffattr;
end;
```

G1.2.5.8 DecodeShareability

```
// DecodeShareability()
// =====
// Decode shareability of target memory region

func DecodeShareability(sh : bits(2)) => Shareability
begin
    case sh of
        when '10' => return Shareability_OSH;
        when '11' => return Shareability_ISH;
        when '00' => return Shareability_NSH;
        otherwise =>
            case ConstrainUnpredictable(Unpredictable_Shareability) of
                when Constraint_OSH => return Shareability_OSH;
                when Constraint_ISH => return Shareability_ISH;
            end;
    end;
end;
```

```
        when Constraint_NSH => return Shareability_NSH;
    end;
end;
end;
```

G1.2.5.9 EffectiveShareability

```
// EffectiveShareability()
// =====
// Force Outer Shareability on Device and Normal iNCoNC memory

func EffectiveShareability(memattrs : MemoryAttributes) => Shareability
begin
    if (memattrs.memtype == MemType_Device ||
        (memattrs.inner.attrs == MemAttr_NC &&
         memattrs.outer.attrs == MemAttr_NC)) then
        return Shareability_OSH;
    else
        return memattrs.shareability;
    end;
end;
```

G1.2.5.10 IsWBShareable

```
// IsWBShareable()
// =====
// Determines whether the given memory attributes are iWBoWB Shareable

func IsWBShareable(memattrs : MemoryAttributes) => boolean
begin

    return (memattrs.memtype == MemType_Normal &&
            memattrs.inner.attrs == MemAttr_WB &&
            memattrs.outer.attrs == MemAttr_WB &&
            memattrs.shareability IN {Shareability_ISH, Shareability_OSH});

end;
```

G1.2.5.11 NormalNCMemAttr

```
// NormalNCMemAttr()
// =====
// Normal Non-cacheable memory attributes

func NormalNCMemAttr() => MemoryAttributes
begin
    var non_cacheable : MemAttrHints;
    non_cacheable.attrs = MemAttr_NC;

    var nc_memattrs : MemoryAttributes;
    nc_memattrs.memtype      = MemType_Normal;
    nc_memattrs.outer        = non_cacheable;
    nc_memattrs.inner        = non_cacheable;
    nc_memattrs.shareability = Shareability_OSH;

    return nc_memattrs;
end;
```

G1.2.5.12 S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =====
// Determine whether a reserved value occupies MAIR_ELx.AttrN

func S1ConstrainUnpredictableRESMAIR(attr : bits(8), slaarch64 : boolean) => boolean
begin
    case attr of
        when '0000xxxx' => return attr[1:0] != '00';
        when 'xxxx0000' => return TRUE;
        otherwise      => return FALSE;
    end;
end;
```

G1.2.5.13 S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1

func S1DecodeMemAttrs(attr_in : bits(8), sh : bits(2), slaarch64 : boolean) => MemoryAttributes
begin
    var attr : bits(8) = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, slaarch64) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits{8}(Unpredictable_RESMAIR);
    end;

    var memattrs : MemoryAttributes;
    case attr of
        when '0000xxxx' => // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device  = DecodeDevice(attr[3:2]);
        otherwise =>
            memattrs.memtype = MemType_Normal;
            memattrs.outer   = DecodeLDFAttr(attr[7:4]);
            memattrs.inner   = DecodeLDFAttr(attr[3:0]);

    end;

    memattrs.shareability = DecodeShareability(sh);

    return memattrs;
end;
```

G1.2.5.14 S2CombineS1AttrHints

```
// S2CombineS1AttrHints()
// =====
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

func S2CombineS1AttrHints(s1_attrhints : MemAttrHints, s2_attrhints : MemAttrHints) => MemAttrHints
begin
    var attrhints : MemAttrHints;

    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;
    end;

    // Stage 2 does not assign any allocation hints
```

```
// Instead, they are inherited from stage 1
if attrhints.attrs != MemAttr_NC then
    attrhints.hints = s1_attrhints.hints;
    attrhints.transient = s1_attrhints.transient;
end;

return attrhints;
end;
```

G1.2.5.15 S2CombineS1Device

```
// S2CombineS1Device()
// =====
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

func S2CombineS1Device(s1_device : DeviceType, s2_device : DeviceType) => DeviceType
begin
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;
    end;
end;
```

G1.2.5.16 S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====
// Combine stage 2 with stage 1 memory attributes

func S2CombineS1MemAttrs(s1_memattrs : MemoryAttributes, s2_memattrs : MemoryAttributes,
                        s2aarch64 : boolean) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;

    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.memtype = MemType_Device;
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elsif s1_memattrs.memtype == MemType_Device then // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elsif s2_memattrs.memtype == MemType_Device then // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);
    end;

    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
end;
```

G1.2.5.17 S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =====
// Combine stage 2 shareability with stage 1

func S2CombineS1Shareability(s1_shareability : Shareability,
                             s2_shareability : Shareability) => Shareability
begin

    if (s1_shareability == Shareability_OSH ||
        s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH ||
           s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
    end;
end;
```

G1.2.5.18 S2DecodeCacheability

```
// S2DecodeCacheability()
// =====
// Determine the stage 2 cacheability for Normal memory

func S2DecodeCacheability(attr : bits(2)) => MemAttrHints
begin
    var s2attr : MemAttrHints;

    case attr of
        when '01' => s2attr.attrs = MemAttr_NC; // Non-cacheable
        when '10' => s2attr.attrs = MemAttr_WT; // Write-through
        when '11' => s2attr.attrs = MemAttr_WB; // Write-back
        otherwise => unreachable;
    end;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints      = ARBITRARY : bits(2);
    s2attr.transient = ARBITRARY : boolean;

    return s2attr;
end;
```

G1.2.5.19 S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// =====
// Decode stage 2 memory attributes

func S2DecodeMemAttrs(attr_in : bits(4), sh : bits(2), s2aarch64 : boolean) => MemoryAttributes
begin
    var memattrs : MemoryAttributes;

    var attr : bits(4);
    if S2ResMemAttr(s2aarch64, attr_in) then
        // Map reserved encodings to an allocated encoding
        (attr, sh) = ConstrainUnpredictableBits{4}(Unpredictable_S2RESMEMATTR);
    else
        attr = attr_in;
    end;

    case attr of
        when '00xx' => // Device memory
```

```
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DecodeDevice(attr[1:0]);
    otherwise => // Normal memory
        memattrs.memtype      = MemType_Normal;
        memattrs.outer        = S2DecodeCacheability(attr[3:2]);
        memattrs.inner        = S2DecodeCacheability(attr[1:0]);
    end;

    memattrs.shareability = DecodeShareability(sh);

    return memattrs;
end;
```

G1.2.5.20 S2ResMemAttr

```
// S2ResMemAttr()
// =====
// Determine whether a reserved value occupies stage 2 MemAttr field

func S2ResMemAttr(s2aarch64 : boolean, attr : bits(4)) => boolean
begin
    case attr of
        when '0100' => return TRUE;
        when '1x00' => return TRUE;
        otherwise   => return FALSE;
    end;
end;
```

G1.2.5.21 AlignmentFault

```
// AlignmentFault()
// =====
// Return a fault record indicating an Alignment fault not due to memory type has occurred
// for a specific access

func AlignmentFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, vaddress);
    fault.statuscode        = Fault_Alignment;

    return fault;
end;
```

G1.2.5.22 ExclusiveFault

```
// ExclusiveFault()
// =====
// Return a fault record indicating a fault for an unsupported Exclusive access

func ExclusiveFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault(accdesc, vaddress);
    fault.statuscode        = Fault_Exclusive;

    return fault;
end;
```

G1.2.5.23 NoFault

```
// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred

func NoFault() => FaultRecord
begin
    var fault : FaultRecord;

    fault.vaddress = ARBITRARY : bits(64);
    fault.statuscode = Fault_None;
    fault.accessdesc = ARBITRARY : AccessDescriptor;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    return fault;
end;

// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred for a specific access

func NoFault(accdesc : AccessDescriptor) => FaultRecord
begin
    var fault : FaultRecord;

    fault.statuscode = Fault_None;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.write = !accdesc.read && accdesc.write;

    return fault;
end;

// NoFault()
// =====

func NoFault(accdesc : AccessDescriptor, vaddress : bits(64)) => FaultRecord
begin
    var fault : FaultRecord = NoFault();
    fault.accessdesc = accdesc;
    fault.write = !accdesc.read && accdesc.write;
    fault.vaddress = vaddress;

    return fault;
end;
```

G1.2.5.24 S1TranslationRegime

```
// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

readonly func S1TranslationRegime(el : bits(2)) => bits(2)
begin
    if el != EL0 then
        return el;

    else
        return EL1;
    end;
end;

// S1TranslationRegime()
```

```
// =====  
// Returns the Exception level controlling the current Stage 1 translation regime. For the most  
// part this is unused in code because the System register accessors (SCTLR_ELx, etc.) implicitly  
// return the correct value.  
  
readonly func S1TranslationRegime() => bits(2)  
begin  
    return S1TranslationRegime(PSTATE.EL);  
end;
```

G1.2.5.25 AddressDescriptor

```
// AddressDescriptor  
// =====  
// Descriptor used to access the underlying memory array.  
  
type AddressDescriptor of record {  
    fault : FaultRecord, // fault.statuscode indicates whether the address is valid  
    memattrs : MemoryAttributes,  
    paddress : FullAddress,  
    vaddress : bits(64)  
};  
  
constant FINAL_LEVEL : integer = 3;
```

G1.2.5.26 CreateAddressDescriptor

```
// CreateAddressDescriptor()  
// =====  
// Set internal members for address descriptor type to valid values  
  
func CreateAddressDescriptor(pa : FullAddress, memattrs : MemoryAttributes,  
                             accdesc : AccessDescriptor) => AddressDescriptor  
begin  
    var addrdesc : AddressDescriptor;  
  
    addrdesc.paddress = pa;  
    addrdesc.memattrs = memattrs;  
    addrdesc.fault = NoFault(accdesc);  
  
    return addrdesc;  
end;  
  
// CreateAddressDescriptor()  
// =====  
// Set internal members for address descriptor type to valid values  
  
func CreateAddressDescriptor(va : bits(64), pa : FullAddress,  
                             memattrs : MemoryAttributes,  
                             accdesc : AccessDescriptor) => AddressDescriptor  
begin  
    var addrdesc : AddressDescriptor;  
  
    addrdesc.paddress = pa;  
    addrdesc.vaddress = va;  
    addrdesc.memattrs = memattrs;  
    addrdesc.fault = NoFault(accdesc);  
  
    return addrdesc;  
end;
```

G1.2.5.27 CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// =====
// Set internal members for address descriptor type with values indicating error

func CreateFaultyAddressDescriptor(fault : FaultRecord) => AddressDescriptor
begin
    var addrdesc : AddressDescriptor;

    addrdesc.vaddress = fault.vaddress;
    addrdesc.fault    = fault;

    return addrdesc;
end;
```

G1.2.5.28 HasUnprivileged

```
// HasUnprivileged()
// =====
// Returns whether a translation regime serves EL0 as well as a higher EL

readonly func HasUnprivileged(regime : Regime) => boolean
begin
    return (regime IN {
        Regime_EL10
    });
end;
```

G1.2.5.29 NUM_PABITS

```
// NUM_PABITS
// =====

constant NUM_PABITS : integer{} = 48;
```

G1.2.5.30 Regime

```
// Regime
// =====
// Translation regimes

type Regime of enumeration {
    Regime_EL2,          // EL2
    Regime_EL10         // EL1&0
};
```

G1.2.5.31 TLBContext

```
// TLBContext
// =====
// Translation context compared on TLB lookups and invalidations, promoting a TLB hit on match

type TLBContext of record {
    ss : SecurityState,
    regime : Regime,
    vmid : bits(NUM_VMIDBITS),
    asid : bits(NUM_ASIDBITS),
    nG : bit,
    ipaspace : PASpace,    // Used in stage 2 lookups & invalidations only
    includes_s1 : boolean,
    includes_s2 : boolean,
    use_vmid : boolean,
    ia : bits(64),        // Input Address
};
```

G1.2.5.32 TLBRecord

```
// TLBRecord
// =====
// Translation output as a TLB payload

type TLBRecord of record {
    blocksize : AddressSize, // Number of bits directly mapped from IA to OA
    contigsize : integer,    // Number of entries log 2 marking a contiguous output range
    s1descriptor : bits(64), // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    s2descriptor : bits(64)  // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
};
```

G1.2.5.33 TranslationRegime

```
// TranslationRegime()
// =====
// Select the translation regime given the target EL and PE state

func TranslationRegime(el : bits(2)) => Regime
begin
    if el == EL2 then
        return Regime_EL2;
    elsif el == EL1 then
        return Regime_EL10;
    elsif el == EL0 then
        return Regime_EL10;
    else
        unreachable;
    end;
end;
```

Part H
Appendixes

Appendix H1

Architectural Constraints on UNPREDICTABLE Behaviors

This chapter describes the architectural constraints on UNPREDICTABLE behaviors in the Armv8 and later architectures. It contains the following sections:

- [AArch32 CONSTRAINED UNPREDICTABLE behaviors](#)

H1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors

From the introduction of Armv8, the architecture defines architecturally required constraints on many behaviors that are UNPREDICTABLE in Armv7. The following sections define those constraints:

- [Overview of the constraints on Armv7 UNPREDICTABLE behaviors.](#)
- [Using R13 by instruction.](#)
- [Using R15 by instruction.](#)
- [Branching into an IT block.](#)
- [Branching to an unaligned PC.](#)
- [Loads and Stores to unaligned locations.](#)
- [CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT.](#)
- [Unallocated System register access instructions.](#)
- [SBZ or SBO fields T32 and A32 in instructions.](#)
- [UNPREDICTABLE cases in immediate constants in T32 data-processing instructions.](#)
- [UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions.](#)
- [CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.](#)
- [CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization](#)
- [Handling of System register control fields for Advanced SIMD and floating-point operation.](#)
- [Mapping of non-idempotent memory locations using the Normal memory type.](#)
- [The Performance Monitors Extension.](#)
- [Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED.](#)
- [Out of range VA.](#)
- [Instruction fetches from Device memory.](#)
- [Multi-access instructions that load the PC from Device memory.](#)
- [Programming CSSELR.Level for a cache level that is not implemented.](#)
- [Crossing a page boundary with different memory types or Shareability attributes.](#)
- [Crossing a 4KB boundary with a Device access.](#)
- [UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs.](#)
- [CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings.](#)
- [Out of range values of the Set/Way/Index fields in cache maintenance instructions.](#)
- [CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set.](#)
- [CONSTRAINED UNPREDICTABLE behavior, T32 and A32 Advanced SIMD and floating-point instructions.](#)
- [CONSTRAINED UNPREDICTABLE behavior of EL2 features.](#)
- [Reserved values in System and memory-mapped registers and translation table entries.](#)
- [CONSTRAINED UNPREDICTABLE behavior in Debug state.](#)

H1.1.1 Overview of the constraints on Armv7 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch32 state, where previous versions of the architecture define behavior as UNPREDICTABLE, from the introduction of Armv8-R the architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note

Software designed to be compatible with the Armv8-R and later architectures must not rely on these CONSTRAINED UNPREDICTABLE cases.

H1.1.2 Using R13 by instruction

In prior versions of the architecture, the use of R13 by instruction as a named register specifier was described as UNPREDICTABLE in the pseudocode. From the introduction of the Armv8-R architecture, the use of R13 as a named register specifier is not UNPREDICTABLE, unless this is specifically stated, and R13 can be used in the regular form. Bits[1:0] of R13 are not treated as SBZP in the Armv7 architecture or RES0 in the Armv8 and later architectures, but can hold any values programmed into them.

H1.1.3 Using R15 by instruction

All uses of R15 by instruction as a named register specifier for a source register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this Manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a NOP.
- Read the Program Counter with the standard offset that applies for the current instruction set.
- Read the Program Counter with the standard offset that applies for the current instruction set with alignment to a word boundary.
- Read 0. This is Arm preferred behavior.
- Read or return an UNKNOWN value for the source register specified as R15.

All uses of R15 as a named register specifier for a destination register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a NOP.
- Ignore the write.
- Branch to an UNKNOWN location in either A32 or T32 state.

Instructions that are CONSTRAINED UNPREDICTABLE when the base register is R15 and the instruction specifies a writeback of the base register, are treated as having R15 as both a source register and a destination register.

For instructions that have two destination registers, for example `LDRD`, `MRRC`, and many of the multiply instructions, if `Rt`, `Rt2`, `RdLo`, or `RdHi` is R15, then the other destination register of the pair is UNKNOWN if the CONSTRAINED UNPREDICTABLE behavior for the write to R15 is either to ignore the write or to branch to an UNKNOWN location.

For instructions that affect any or all of `PSTATE.{N, Z, C, V}`, `PSTATE.Q`, and `PSTATE.GE` when the register specifier is not R15, any flags affected by an instruction that is CONSTRAINED UNPREDICTABLE when the register specifier is R15 become UNKNOWN.

In addition, for `MRC` instructions that use R15 as the destination register descriptor, and therefore target `APSR_nzcv` where these are described as being CONSTRAINED UNPREDICTABLE, `PSTATE.{N, Z, C, V}` becomes UNKNOWN.

H1.1.4 Branching into an IT block

Branching into an IT block leads to CONSTRAINED UNPREDICTABLE behavior. Execution starts from the address determined by the branch, but each instruction in the IT block is:

- Executed as if it were not in an IT block. This means that it is executed unconditionally.
- Executed as if it had passed its [Condition code check](#) within an IT block.
- Executed as a NOP. That is, it behaves as if it had failed the [Condition code check](#).

H1.1.5 Branching to an unaligned PC

In A32 state, when branching to an address that is not word-aligned and is defined to be CONSTRAINED UNPREDICTABLE, one of the following behaviors must occur:

- The unaligned location is forced to be aligned.

- The unaligned address generates a Prefetch Abort on the first instruction using the unaligned PC value. If that instruction is executed at EL0 and [HCR.TGE](#) is 1 the exception is taken to EL2. If the instruction is executed at EL0 when the applicable TGE bit is 0 the exception is taken to EL1. If the instruction is executed at an Exception level that is higher than EL0 the exception is taken to the Exception level at which the instruction was executed. In all cases, the exception is generated only if the first instruction using the unaligned PC value is architecturally executed.

If the exception that results from a branch to an unaligned PC value it is reported as a Prefetch Abort exception, see [Prefetch Abort exception reporting a PC alignment fault exception](#).

Note

Because bit[0] is used for interworking, it is impossible to specify a branch to A32 state when the bottom bit of the target address is 1. Therefore the bottom bit of [IFAR](#), [HIFAR](#), or [FAR_ELx](#) is 0 for all these cases.

H1.1.6 Loads and Stores to unaligned locations

Some unaligned loads and stores in the Armv8-A and later architectures are described as CONSTRAINED UNPREDICTABLE to do one of the following:

- Take an alignment fault.
- Perform the specified load or store to the unaligned memory location.

H1.1.7 CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT

A number of instructions in the architecture are described as being CONSTRAINED UNPREDICTABLE either:

- Anywhere within an IT block.
- As an instruction within an IT block, other than the last instruction within an IT block.

Unless otherwise stated in this manual, when these instructions are committed for execution, one of the following occurs:

- An UNDEFINED exception results.
- The instructions are executed as if they had passed the [Condition code check](#).
- The instructions execute as NOPs. This means that they behave as if they had failed the [Condition code check](#).

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

Many instructions that are CONSTRAINED UNPREDICTABLE in an IT block are branch instructions or other non-sequential instructions that change the PC. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the [PSTATE.IT](#) state machine must be treated in one of the following ways:

- [PSTATE.IT](#) is cleared to 0.
- [PSTATE.IT](#) advances for either a sequential or a nonsequential change of the PC in the same way as it does for instructions that are not CONSTRAINED UNPREDICTABLE that cause a sequential change of the PC.

Note

This does not apply to an instruction that is the last instruction in an IT block.

The instructions addressed by the updated PC must do one of the following:

- Execute as if they had passed the [Condition code check](#) for the remaining iterations of the [PSTATE.IT](#) state machine.

- Execute as NOPs. That is, they behave as if they had failed the [Condition code check](#) for the remaining iterations of the [PSTATE.IT](#) state machine.
- Execute as if they were unconditional, or, if the instructions are part of another IT block, in accordance with the behavior described in [Branching into an IT block](#).

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

For exception returns or Debug state exits that cause [PSTATE.IT](#) to be set to a reserved value in T32 state or that return to A32 state with a nonzero value in [PSTATE.IT](#), the [PSTATE.IT](#) bits are forced to 0b00000000. The reserved values are:

```
PSTATE.IT[7:4] != '0000' && PSTATE.IT[3:0] == '0000'
PSTATE.IT[2:0] != '000' && (SCTLR().ITD == '1' || SCTLR_ELx().ITD == '1')
```

Exception returns or Debug state exits that set [PSTATE.IT](#) to a non-reserved value in T32 state can occur when the flow of execution returns to a point:

- Outside an IT block, but with the [PSTATE.IT](#) bits set to a value other than 0b00000000.
- Inside an IT block, but with a different value of the [PSTATE.IT](#) bits than if the IT block had been executed without an exception return or Debug state exit.

In this case, the instructions at the target of the exception return or Debug state exit must do one of the following:

- Execute as if they passed the [Condition code check](#) for the remaining iterations of the [PSTATE.IT](#) state machine.
- Execute as NOPs. That is, they behave as if they failed the [Condition code check](#) for the remaining iterations of the [PSTATE.IT](#) state machine.
- Execute as if they were unconditional, or as if the instruction were part of another IT block, in accordance with the behavior in [Branching into an IT block](#).

The remaining iterations of the [PSTATE.IT](#) state machine must behave in one of the following ways:

- The [PSTATE.IT](#) state machine advances as if it were in an IT block.
- The [PSTATE.IT](#) bits are ignored.
- The [PSTATE.IT](#) bits are forced to 0b00000000.

H1.1.8 Unallocated System register access instructions

From the introduction of the Armv8-R architecture, accesses to unallocated System register encodings are UNDEFINED.

This includes:

- Reads using encodings that are defined as WO.
- Writes using encodings that are defined as RO.
- MCR or MRC accesses to using a set of {coproc, CRn, opc1, CRm, opc2} values that the Armv7 architecture defined as UNPREDICTABLE.
- MCRR and MRRC instructions with unallocated values of opc1 or CRm that are described as UNPREDICTABLE are UNDEFINED in the Armv8-A and later architectures.

H1.1.9 SBZ or SBO fields T32 and A32 in instructions

Many of the T32 and A32 instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO. If the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.

- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.

The exceptions to this rule are:

- LDM, LDMIA, LDMFD.
- LDMDB, LDMEA.
- LDR (literal).
- LDRB (literal).
- LDRD (immediate).
- LDRD (register).
- LDRD (literal).
- LDRH (literal).
- LDRSB (literal).
- LDRSH (literal).
- POP.
- PUSH.
- SDIV.
- STM, STMIA, STMEA.
- STMDB, STMFD.
- UDIV.

H1.1.10 UNPREDICTABLE cases in immediate constants in T32 data-processing instructions

The description of immediate constants in T32 data processing [Modified immediate constants in T32 instructions](#) include constant values that were UNPREDICTABLE in Armv7. [Instruction encodings](#) describes 32-bit T32 instructions as {hw1, hw2}, where hw1 is the left-hand halfword in the 32-bit encoding diagram for the instruction. The UNPREDICTABLE cases are those where both:

- $hw2[7:0] == 0b0000000$.
- $hw1[10] == 0$ and either:
 - $hw2[14:12] == 0b001$.
 - $hw2[14:12] == 0b010$.
 - $hw2[14:12] == 0b011$.

From the introduction of the Armv8 architecture, the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce the value $0b0000000$.

H1.1.11 UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions

The description of immediate constants in [Modified immediate constants in T32 and A32 Advanced SIMD instructions](#) include constant values that were UNPREDICTABLE in Armv7. The UNPREDICTABLE cases are those where:

- The bits that the encoding diagram shows as *abcd* are all 0.
In the A32 encoding these are bits[24, 18:6, 3:0]. In the T32 encoding they are bits {hw1[12, 2:0], hw2[3:0]}.
- The bits that the encoding diagram shows as *cmode*[3:1] are one of {0b001, 0b010, 0b011, 0b101, 0b110}.
In the A32 encoding these are bits[11:9]. In the T32 encoding they are bits hw2[11:9].

From the introduction of the Armv8 architecture, the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce an immediate constant value of zero.

H1.1.12 CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values

The Arm architecture allows copies of System register control or data values to be cached in a cache or TLB. This can lead to CONSTRAINED UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

H1.1.13 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to CONSTRAINED UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

H1.1.14 Unallocated values with register fields of CP15 registers and Translation Table entries

Unless stated elsewhere, all unallocated or reserved values of fields with allocated values within CP15 registers and Translation Table entries be have in one in one of the following ways:

- The encoding maps onto any of the allocated values but otherwise does not cause UNPREDICTABLE behavior.
- The encodings cause effects that could be achieved by a combination of more than one of the allocated encodings.
- The encodings cause the field to have no functional effect.

H1.1.15 Handling of System register control fields for Advanced SIMD and floating-point operation

For historical reasons described in [Background to the System register interface](#), each of the [CPACR](#), [HCPTR](#), and [NSACR](#) has a pair of control fields that were defined to have identical functionality for controlling Advanced SIMD and floating-point operation. These fields are:

- [CPACR](#).{cp10, cp11}.
- [HCPTR](#).{TCP10, TCP11}.
- [NSACR](#).{cp10, cp11}.

The architecture requires that both fields in one of these pairs are programmed to the same value. If this is not done, then the CONSTRAINED UNPREDICTABLE behavior is that behavior is the same as if the cp11 or TCP11 control field was equal to the cp10 or TCP10 field. This is in all respects except for the value read back by a direct read of the register.

After a register write that writes different values to the two fields of a pair, a direct read of the register might return an UNKNOWN value for the cp11 or TCP11 field.

Note

This means that, when different values are written to the {cp10, cp11} fields in a single register, the architecture permits but does not require that a read of that register returns the value written to the cp11 field.

H1.1.15.1 CONSTRAINED UNPREDICTABLE CPACR and NSACR settings

If [CPACR](#).cp<n> contains the encoding '10', then one of the following behaviors must occur:

- The encoding maps onto any of the allocated values, but otherwise does not cause UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.

Note

In Armv7, [CPACR](#) had a D32DIS bit, and [NSACR](#) had an NSD32DIS bit. There is no [CPACR](#).D32DIS or [NSACR](#).NSD32DIS in Armv8-A and later architectures, and the corresponding bits in the two registers are RES0.

H1.1.16 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in the following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

H1.1.17 The Performance Monitors Extension

I_{WKXBB}

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch32 state:

- [CONSTRAINED UNPREDICTABLE](#) accesses to [PMXEVTYPER](#) or [PMXEVCNTR](#).
- [CONSTRAINED UNPREDICTABLE](#) accesses to [PMEVCNTR](#)<n> and [PMEVTYPER](#)<n>.
- [CONSTRAINED UNPREDICTABLE](#) behavior caused by [HDCR](#).HPMN.

H1.1.17.1 CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR

R_{XSVNC}

If [PMSELR](#).SEL is greater than the number of event counters accessible at the current Exception level, then accesses to [PMXEVTYPER](#) or [PMXEVCNTR](#) can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when any of the following is true:

- If [PMSELR](#).SEL is not equal to 31, and [PMSELR](#).SEL is greater than or equal to [PMCR](#).N, and the PE is executing in EL2.

Note

If EL2 is implemented and enabled in the current Security state, [HDCR](#).HPMN, or [MDCR](#).EL2.HPMN, identifies the number of accessible counters at EL0 or EL1. Otherwise, the number of accessible counters is the number of accessible event counters.

Accesses from EL0 to [PMXEVCNTR](#) are permitted when the values of [PMUSERENR](#).{ER, EN} are both 1.

Accesses from EL0 to [PMXEVTYPER](#) are permitted when value of [PMUSERENR](#).EN is 1.

I_{HNSSP}

If [PMSELR](#).SEL is equal to 31, the PE does one of the following:

- Accesses to [PMXEVCNTR](#) are UNDEFINED.
- Accesses to [PMXEVCNTR](#) behave as RAZ/WI.
- Accesses to [PMXEVCNTR](#) execute as NOPs.
- Accesses to [PMXEVCNTR](#) behave as if [PMSELR](#).SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- If an access to [PMXEVCNTR](#) from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

I_{GRBHK}

H1.1.17.2 CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>

If <n> is greater than the number of event counters available in the current Exception level and state, reads and writes of [PMEVCNTR<n>](#) and [PMEVTYPER<n>](#) are CONSTRAINED UNPREDICTABLE, and the PE does one of the following:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- If EL2 is implemented and enabled in the current Security state, for an access to [PMEVCNTR<n>](#) or [PMEVTYPER<n>](#) from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.
Accesses from EL0 are permitted to [PMEVTYPER<n>](#) when the value of [PMUSERENR.EN](#) is 1.

Note

At EL0 and EL1 [HDCR.HPMN](#) identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of accessible event counters.

R_{ZLPGP}

H1.1.17.3 CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN

If NUM_PMU_COUNTERS is nonzero, and [HDCR.HPMN](#) is 0, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of [HDCR.HPMN](#) is UNKNOWN.
- Either:
 - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if [HDCR.HPMN](#) is set to an UNKNOWN nonzero value less than or equal to NUM_PMU_COUNTERS.
 - All counters are reserved for EL2 use, meaning no counters are accessible from EL1 and EL0.

H1.1.18 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, this generates an exception:

- If this exception is taken to an Exception level that is using AArch64, then ESR_ELx is UNKNOWN.
- If this exception is taken to EL2 and EL2 is using AArch32, then the [HSR](#) is unknown.

Note

The value written to ESR or HSR must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but resulted from a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

H1.1.19 Out of range VA

If the PE executes an instruction for which the instruction address, size, and alignment mean it contains the bytes 0xFFFF_FFFF and 0x0000_0000, then the bytes that wrap around and appear to be from 0x0000_0000 onwards come from an UNKNOWN address.

If the PE executes a load or store instruction for which the calculated address, total access size, and alignment mean it accesses bytes 0xFFFF_FFFF and 0x0000_0000, then the bytes that wrap around and appear to be from 0x0000_0000 onwards come from an UNKNOWN address.

H1.1.20 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location when address translation is enabled.

If a branch causes the Program Counter to point to a location in memory with the Device attribute that is not marked as execute-never for the current Exception level for instruction fetches, then an implementation must perform one of the following behaviors:

- It treats the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- It generates a Permission fault.

H1.1.21 Multi-access instructions that load the PC from Device memory

Multi-access instructions that load the PC from Device memory when address translation is enabled are UNPREDICTABLE in AArch32 state. From the introduction of the Armv8-A architecture in AArch32 state an implementation must perform one of the following behaviors:

- It loads the PC from the memory location as if the memory location had the Normal Non-cacheable attribute.
- It generates a Permission fault.

H1.1.22 Programming CSSELR.Level for a cache level that is not implemented

If CSSELR.Level is programmed to a cache level that is not implemented, then a read of CSSELR returns an UNKNOWN value in CSSELR.Level.

If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR an implementation must perform one of the following behaviors:

- The CCSIDR read is treated as a NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

H1.1.23 Crossing a page boundary with different memory types or Shareability attributes

If a single load or store instruction generates multiple memory accesses, such that the total set of accesses crosses a page boundary to a memory location that has a different memory type, Normal or Device, or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an alignment fault caused by the memory type.
For the Non-secure PL1&0 translation regime:
 - If the stage 1 translation causes the mismatch, the resulting exception is taken to PL1.
 - If the stage 2 translation causes the mismatch, the resulting exception is taken to PL2.
 - If both stages of translation cause the mismatch, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

H1.1.24 Crossing a 4KB boundary with a Device access

A memory access from a load or store instruction to Device memory that crosses a 4KB boundary results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses, except that there is no guarantee of ordering between memory accesses.
- The instruction generates an Alignment fault caused by the memory type.
For the Non-secure PL1&0 translation regime:
 - If the stage 1 translation causes the boundary to be crossed, the resulting exception is taken to PL1.
 - If the stage 2 translation causes the boundary to be crossed, the resulting exception is taken to PL2.

- If both stages of translation cause the boundary to be crossed, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

Note

The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.

H1.1.25 UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

[Load-Exclusive and Store-Exclusive instruction usage restrictions](#) defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. These cases were UNPREDICTABLE in Armv7. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
- The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:
 - Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
 - Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in [Load-Exclusive and Store-Exclusive instruction usage restrictions](#) for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

Note

Additional CONSTRAINED UNPREDICTABLE cases can apply to Load-Exclusive and Store-Exclusive instructions, see [CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set](#).

H1.1.26 CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings

The T32 and A32 instruction sets include encodings that result in CONSTRAINED UNPREDICTABLE behavior when they are decoded.

H1.1.26.1 CONSTRAINED UNPREDICTABLE behavior of CRC32 instruction encodings

In the T32 and A32 instruction sets, there are encodings of the CRC32 and CRC32C instructions that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in the following places in the T32 and A32 instruction sets:

- [Cyclic Redundancy Check](#) for the A32 instruction set, with `sz = 11`.
- [Data-processing \(two source registers\)](#) for the T32 instruction set, with `op1 = 10x` and `op2 = 11`.

The CONSTRAINED UNPREDICTABLE behavior for these encodings is described in CRC32 and CRC32C.

H1.1.26.2 CONSTRAINED UNPREDICTABLE behavior of other A32 instruction encodings

In the A32 instruction set, there are encodings that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in:

- [Miscellaneous](#).
- [Memory hints and barriers](#).

- [Barriers.](#)

The CONSTRAINED UNPREDICTABLE behavior is that an implementation must treat the encodings in one of the following ways:

- The instruction encoding is UNDEFINED.
- The instruction encoding executes as a NOP.

H1.1.27 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions [DCCISW](#), [DCCSW](#), and [DCISW](#), if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
 - No cache lines.
 - A single arbitrary cache line.
 - Multiple arbitrary cache lines.

H1.1.28 CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different T32 and A32 System instructions.

Note

If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

H1.1.28.1 SRS (T32)

For a description of this instruction and the encoding, see [SRS](#), [SRSDA](#), [SRSDb](#), [SRSIA](#), [SRSIB](#).

H1.1.28.1.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
 - The instruction is UNDEFINED.
 - The instruction executes as a NOP.
 - R13 of the current mode is used.
 - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

H1.1.28.2 SRS (A32)

For a description of this instruction and the encoding, see [SRS](#), [SRSDA](#), [SRSDb](#), [SRSIA](#), [SRSIB](#).

H1.1.28.2.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
 - The instruction is UNDEFINED.
 - The instruction executes as a NOP.
 - R13 of the current mode is used.
 - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

H1.1.28.3 SUBS PC, LR and related instructions (T32)

For a description of this instruction and the encoding, see the exception return form of [SUB, SUBS \(immediate\)](#).

H1.1.28.3.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
 - The instruction is UNDEFINED.
 - The instruction executes as a NOP.
- If the instruction transfers an illegal mode encoding to [PSTATE.M](#), then this invokes the illegal exception return.

Note

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

For encoding T5:

- If `hw1[3:0]` are not `0b1110`, and the instruction is executed when not in Hyp mode, System mode, or User mode, then one of the following behaviors must occur:
 - The instruction is UNDEFINED.
 - The instruction is treated as a NOP.
 - The instruction is treated as if `hw1[3:0]` are `0b1110`.
 - The Program Counter is set using the value in the register specified by `hw1[3:0]`.

H1.1.28.4 SUBS PC, LR and related instructions (A32)

For a description of this instruction and the encoding, see the exception return forms of [MOV, MOVS \(register\)](#) and [SUB, SUBS \(immediate\)](#).

H1.1.28.4.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
 - The instruction is UNDEFINED.
 - The instruction executes as a NOP.
- If the instruction transfers an illegal mode encoding to [PSTATE.M](#), then this invokes the illegal exception return.

Note

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

H1.1.29 CONSTRAINED UNPREDICTABLE behavior, T32 and A32 Advanced SIMD and floating-point instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different T32 and A32 Advanced SIMD and floating-point instructions listed in [Alphabetical list of Advanced SIMD and floating-point instructions](#).

Note

- The pseudocode used in this section to describe cases that can result in CONSTRAINED UNPREDICTABLE behavior does not necessarily match the encoding-specific pseudocode for a specific instruction.
 - If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.
-

H1.1.29.1 VCVT (between floating-point and fixed-point)

For a description of this instruction and the encoding, see [VCVT \(between floating-point and fixed-point, floating-point\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.2 VLD1 (multiple single elements)

For a description of this instruction and the encoding, see [VLD1 \(multiple single elements\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.3 VLD1 (single element to all lanes)

For a description of this instruction and the encoding, see [VLD1 \(single element to all lanes\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.4 VLD2 (multiple 2-element structures)

For a description of this instruction and the encoding, see [VLD2 \(multiple 2-element structures\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.5 VLD2 (single 2-element structure to one lane)

For a description of this instruction and the encoding, see [VLD2 \(single 2-element structure to one lane\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.6 VLD2 (single 2-element structure to all lanes)

For a description of this instruction and the encoding, see [VLD2 \(single 2-element structure to all lanes\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.7 VLD3 (multiple 3-element structures)

For a description of this instruction and the encoding, see [VLD3 \(multiple 3-element structures\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.8 VLD3 (single 3-element structure to one lane)

For a description of this instruction and the encoding, see [VLD3 \(single 3-element structure to one lane\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.9 VLD3 (single 3-element structure to all lanes)

For a description of this instruction and the encoding, see [VLD3 \(single 3-element structure to all lanes\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.10 VLD4 (multiple 4-element structures)

For a description of this instruction and the encoding, see [VLD4 \(multiple 4-element structures\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.11 VLD4 (single 4-element structure to one lane)

For a description of this instruction and the encoding, see [VLD4 \(single 4-element structure to one lane\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.12 VLD4 (single 4-element structure to all lanes)

For a description of this instruction and the encoding, see [VLD4 \(single 4-element structure to all lanes\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.13 VLDM

For a description of this instruction and the encoding, see [VLDM](#), [VLDMDB](#), [VLDMIA](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.14 VMOV (between two general-purpose registers and two single-precision registers)

For a description of this instruction and the encoding, see [VMOV \(between two general-purpose registers and two single-precision registers\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.15 VMOV (between two general-purpose registers and a doubleword floating-point register)

For a description of this instruction and the encoding, see [VMOV \(between two general-purpose registers and a doubleword floating-point register\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.16 VST1 (multiple single elements)

For a description of this instruction and the encoding, see [VST1 \(multiple single elements\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.17 VST2 (multiple 2-element structures)

For a description of this instruction and the encoding, see [VST2 \(multiple 2-element structures\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.18 VST2 (single 2-element structure from one lane)

For a description of this instruction and the encoding, see [VST2 \(single 2-element structure from one lane\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.19 VST3 (multiple 3-element structures)

For a description of this instruction and the encoding, see [VST3 \(multiple 3-element structures\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.20 VST3 (single 3-element structure from one lane)

For a description of this instruction and the encoding, see [VST3 \(single 3-element structure from one lane\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.21 VST4 (multiple 4-element structures)

For a description of this instruction and the encoding, see [VST4 \(multiple 4-element structures\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.22 VST4 (single 4-element structure from one lane)

For a description of this instruction and the encoding, see [VST4 \(single 4-element structure from one lane\)](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.29.23 VSTM

For a description of this instruction and the encoding, see [VSTM](#), [VSTMDB](#), [VSTMIA](#).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

H1.1.30 CONSTRAINED UNPREDICTABLE behavior of EL2 features

The following sections, and the instruction descriptions, describe CONSTRAINED UNPREDICTABLE behavior that can occur in an implementation that includes EL2 where EL2 can use AArch32:

- [ERET in User mode or System mode](#).
- [Illegal entry into, or exit from, Hyp mode](#).
- [Stage 1 default memory type](#).
- [Trapping of general exceptions to Hyp mode](#).
- [MSR \(banked register\) and MRS \(banked register\)](#).

H1.1.30.1 ERET in User mode or System mode

If ERET is executed in User mode or System mode, it behaves as described in [SUBS PC, LR and related instructions \(T32\)](#).

H1.1.30.2 Illegal entry into, or exit from, Hyp mode

Illegal entry to Hyp mode or exit from Hyp mode might be due to modifying [PSTATE.M](#) when in Hyp mode.

Attempting to change into Hyp mode or out of Hyp mode using the [MSR](#) or [CPS](#) instruction invokes the illegal exception return by not changing the mode, and setting [PSTATE.IL](#) to 1.

SRS using the Hyp mode SP from modes other than Hyp mode is handled as described in [SRS \(T32\)](#) and [SRS \(A32\)](#).

H1.1.30.3 Stage 1 default memory type

If [HCR.DC](#) = 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with:

- [SCTLR.M](#) = 0, regardless of the actual value of [SCTLR.M](#), other than for the value returned by an explicit read of [SCTLR.M](#).
- [HCR.VM](#) = 1, regardless of the actual value of [HCR.VM](#), other than for an explicit read of this bit.

H1.1.30.4 Trapping of general exceptions to Hyp mode

Attempting to perform an exception return to a PL1 mode when `HCR.TGE == 1` invokes an illegal exception return.

Attempting to change from Monitor mode to a PL1 mode when `HCR.TGE == 1` by executing a `CPS` or `MSR` instruction generates an Illegal Execution state exception, by not changing the mode, and setting `PSTATE.IL` to 1.

Because taking an exception into Non-secure PL1 modes leads to a CONSTRAINED UNPREDICTABLE situation, the following additional properties apply when `HCR.TGE == 1`:

- All exceptions that would be routed to EL1 are routed to EL2.
- `SCTLR.M` is treated as being 0, regardless of its actual value, other than for an explicit read of this bit.
- `HCR.FMO`, `HCR.IMO`, and `HCR.AMO` are treated as being 1, regardless of their actual value, other than for an explicit read of these bits.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.

H1.1.30.5 MSR (banked register) and MRS (banked register)

If the target register specified by the `{R, SYSm}` fields of the instruction encoding is not accessible from the PE mode in which the instruction was executed (see [Usage restrictions on the banked register transfer instructions](#)), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For MRS (banked register) instructions, the destination general-purpose register becomes UNKNOWN.
- For MSR (banked register) instructions, if the register specified could be accessed from the current mode by other mechanisms, then this register is UNKNOWN. Otherwise, the instruction is a NOP.

If the instruction was executed specifying an unallocated `{R, SYSm}` field value or an unimplemented register (see [Encoding the register argument in the banked register transfer instructions](#)), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- An allocated MRS (banked register) or MSR (banked register) instruction is executed.

H1.1.31 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated, all unallocated or reserved values of fields with allocated values within the AArch32 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
- The encoding causes the field to have no functional effect.

Note

These constraints are identical to those for the equivalent AArch64 definitions, as given in Reserved values in System and memory-mapped registers and translation table entries.

Unless otherwise stated, when a direct write of a System register or memory-mapped register writes an unallocated value to a field, if that value is unallocated in all contexts then a subsequent read of the field returns an UNKNOWN value.

H1.1.32 CONSTRAINED UNPREDICTABLE behavior in Debug state

[Behavior in Debug state](#) of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

Appendix H2

Recommendations for Reporting Memory Attributes on an Interconnect

This appendix describes the Arm recommendations for reporting the memory attributes that are assigned by the PE. It contains the following section:

- [Arm recommendations for reporting memory attributes on an interconnect](#)

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

The Arm architecture defines the architectural interface between software and the PE hardware. This means the mechanisms by which different memory type and Cacheability attributes are presented on an interface to an interconnect fabric such as AMBA[®] AXI are, strictly, outside the scope of the architecture. This appendix describes an approach for the interface between a PE implementation and an interconnect fabric that Arm strongly recommends, but these recommendations do not form part of the Arm architecture.

H2.1.1 Effect of microarchitectural choices on memory attributes

Implementations of the Arm architecture permit considerable variability in the presentation of memory attributes on the interconnect fabric, particularly in cases where the PE implementation does not provide optimized support for a memory type. For example, an implementation might treat Write-Through locations as Non-cacheable at some level of cache, because functionally this is consistent with the definition of Write-Through, but for the particular implementation the performance trade-off does not merit the hardware directly providing Write-Through capability. However, in such implementations, the assigned memory attributes are not changed by the microarchitectural choices. The microarchitecture simply implements different ways of handling some memory attributes.

Therefore, Arm strongly recommended that where any or all of the following memory attributes are presented on the interface between a PE and an interconnect fabric, the attributes that are presented are completely consistent with the attributes defined by the translation system:

- The memory type, Normal or Device.
- The Early Write Acknowledgment attribute.
- The ordering requirements.
- The Shareability.
- The Cacheability, including where practicable, the allocation hints.

H2.1.1.1 Effect when memory accesses are forced to be Non-cacheable

Arm also strongly recommends that the effects of forcing accesses to Normal memory to be Non-cacheable, as described in Enabling and disabling the caching of memory accesses for AArch64 and in [Enabling and disabling the caching of memory accesses in AArch32 state](#) for AArch32, are reflected on the interconnect by the memory type and attributes used for memory transactions generated while the cache is disabled.

Appendix H3

Legacy Instruction Syntax for AArch32 Instruction Sets

This appendix describes the legacy instruction syntax in the Arm instruction sets, and their *Unified Assembler Language* (UAL) equivalents. It contains the following section:

- [Legacy Instruction Syntax](#)

H3.1 Legacy Instruction Syntax

Early versions of the Arm Architecture defined an assembly language for A32 (ARM) instructions, and a separate assembly language for T32 (Thumb) instructions. UAL is based on the A32 assembly language, with some changes to the instruction syntax. The appendix describes those changes. The pre-UAL mnemonics are compatible with UAL, and might be supported by an assembler.

The original T32 assembly language is not compatible with UAL, and is not described in the manual.

H3.1.1 Pre-UAL instruction syntax for the A32 base instructions

Table H3-1 lists the syntax for the A32 base instructions that have changed after UAL was introduced.

Table H3-1 Pre-UAL instruction syntax for the A32 base instructions

Pre-UAL syntax	UAL equivalent	See
ADC<c>S	ADCS<c>	ADC, ADCS (immediate), ADC, ADCS (register), ADC, ADCS (register-shifted register)
ADD<c>S	ADDS<c>	ADD, ADDS (immediate), ADD, ADDS (register), ADD, ADDS (register-shifted register), ADD, ADDS (SP plus immediate), ADD, ADDS (SP plus register)
AND<c>S	ANDS<c>	AND, ANDS (immediate), AND, ANDS (register), AND, ANDS (register-shifted register)
BIC<c>S	BICS<c>	BIC, BICS (immediate), BIC, BICS (register), BIC, BICS (register-shifted register)
EOR<c>S	EORS<c>	EOR, EORS (immediate), EOR, EORS (register), EOR, EORS (register-shifted register)
LDC<c>L	LDCL<c>	LDC (immediate), LDC (literal)
LDM<c>IA, LDM<c>FD	LDM<c>	LDM, LDMIA, LDMFD
LDM<c>DA, LDM<c>FA	LDMDA<c>	LDMDA, LDMFA
LDM<c>DB, LDM<c>EA	LDMDB<c>	LDMDB, LDMEA
LDM<c>IB, LDM<c>ED	LDMIB<c>	LDMIB, LDMED
LDR<c>B	LDRB<c>	LDRB (immediate), LDRB (literal), LDRB (register)
LDR<c>BT	LDRBT<c>	LDRBT
LDR<c>D	LDRD<c>	LDRD (immediate), LDRD (literal), LDRD (register)
LDR<c>H	LDRH<c>	LDRH (immediate), LDRH (literal), LDRH (register)

Pre-UAL syntax	UAL equivalent	See
LDR<c>SB	LDRSB<c>	LDRSB (immediate), LDRSB (literal), LDRSB (register)
LDR<c>SH	LDRSH<c>	LDRSH (immediate), LDRSH (literal), LDRSH (register)
LDR<c>T	LDRT<c>	LDRT
MLA<c>S	MLAS<c>	MLA, MLAS
LSLS <Rd>, <Rn>, #0	MOVS <Rd>, <Rm>	MOV, MOVS (immediate), MOV, MOVS (register)
MOV<c>S	MOVS<c>	
MUL<c>S	MULS<c>	MUL, MULS
MVN<c>S	MVNS<c>	MVN, MVNS (immediate), MVN, MVNS (register), MVN, MVNS (register-shifted register)
ORR<c>S	ORRS<c>	ORR, ORRS (immediate), ORR, ORRS (register), ORR, ORRS (register-shifted register)
QADDSUBX	QASX	QASX
QSUBADDX	QSAX	QSAX
RSB<c>S	RSBS<c>	RSB, RSBS (immediate), RSB, RSBS (register), RSB, RSBS (register-shifted register)
RSC<c>S	RSCS<c>	RSC, RSCS (immediate), RSC, RSCS (register), RSC, RSCS (register-shifted register)
SADDSUBX	SASX	SASX
SBC<c>S	SBCS<c>	SBC, SBCS (immediate), SBC, SBCS (register), SBC, SBCS (register-shifted register)
SHADDSUBX	SHASX	SHASX
SHSUBADDX	SHSAX	SHSAX
SMI<c>	SMC<c>	SMC
SMLAL<c>S	SMLALS<c>	SMLAL, SMLALS
SMULL<c>S	SMULLS<c>	SMULL, SMULLS
SSUBADDX<c>	SSAX<c>	SSAX
STC<c>L	STCL<c>	STC
STM<c>EA, STM<c>IA	STM<c>	STM, STMIA, STMEA
STM<c>DA, STM<c>ED	STMDA<c>	STMDA, STMED
STM<c>DB, STM<c>FD	STMDB<c>	STMDB, STMFD
STM<c>IB, STM<c>FA	STMIB<c>	STMIB, STMFA
STR<c>B	STRB<c>	STRB (immediate), STRB (register)
STR<c>BT	STRBT<c>	STRBT
STR<c>D	STRD<c>	STRD (immediate), STRD (register)

Pre-UAL syntax	UAL equivalent	See
STR<c>H	STRH<c>	STRH (immediate), STRH (register)
STR<c>T	STRT<c>	STRT
SUB<c>S	SUBS<c>	SUB, SUBS (immediate), SUB, SUBS (register), SUB, SUBS (register-shifted register), SUB, SUBS (SP minus immediate), SUB, SUBS (SP minus register)
SWI	SVC	SVC
UADDSUBX	UASX	UASX
UHADDSUBX	UHASX	UHASX
UHSUBADDX	UHSAX	UHSAX
UMLAL<c>S	UMLALS<c>	UMLAL, UMLALS
UMULL<c>S	UMULLS<c>	UMULL, UMULLS
UQADDSUBX	UQASX	UQASX
UQSUBADDX	UQSAX	UQSAX
USUBADDX	USAX	USAX
UEXT8	UXTB	UXTB
UEXT16	UXTH	UXTH

H3.1.2 Pre-UAL instruction syntax for the A32 floating-point instructions

Table H3-2 lists the syntax for A32 floating-point instructions that have changed after UAL was introduced.

Table H3-2 Pre-UAL instruction syntax for A32 floating-point instructions

Pre-UAL syntax	UAL equivalent	See
FABSD	VABS.F64	VABS
FABSS	VABS.F32	
FADDD	VADD.F64	VADD (floating-point)
FADDS	VADD.F32	
FCMPEZD	VCMPE.F64	VCMPE
FCMPEZS	VCMPE.F32	
FCMPZD	VCMP.F64	VCMP,
FCMPZS	VCMP.F32	
FCONSTD <Dd>, #<imm8>	VMOV.F64 <Dd>, #<fpimm>	VMOV (immediate) For more information, see FCONST.
FCONSTS <Sd>, #<imm8>	VMOV.F32 <Sd>, #<fpimm>	
FCPYD	VMOV.F64	VMOV (register)
FCPYS	VMOV.F32	
FCVTDS	VCVT.F64.F32	VCVT (between double-precision and single-precision)

Pre-UAL syntax	UAL equivalent	See
FCVTS	VCVT.F32.F64	
FDIVD	VDIV.F64	VDIV
FDIVS	VDIV.F32	
FLDD	VLDR.F64	VLDR (immediate) VLDR (literal)
FLDMD, FLDMIAD	VLDM.F64	VLDM, VLDMDB, VLDMIA
FLDMS	VLDM.F32	
FLDS	VLDR.F32	VLDR (immediate) VLDR (literal)
FMACD	VMLA.F64	VMLA (floating-point)
FMACS	VMLA.F32	
FMDHR <Dd>, <Rt>	VMOV <Dd[1]>, <Rt>	VMOV (general-purpose register to scalar)
FMDLR <Dd>, <Rt>	VMOV <Dd[0]>, <Rt>	
FMDRR	VMOV	VMOV (between two general-purpose registers and a doubleword floating-point register)
FMRDH <Rt>, <Dd>	VMOV <Rt>, <Dd[1]>	VMOV (scalar to general-purpose register)
FMRDL <Rt>, <Dd>	VMOV <Rt>, <Dd[0]>	
FMRRD	VMOV	VMOV (between two general-purpose registers and a doubleword floating-point register)
FMRRS	VMOV	VMOV (between two general-purpose registers and two single-precision registers)
FMRS	VMOV	VMOV (between general-purpose register and single-precision)
FMRX	VMRS	VMRS
FMSCD	VNMLS.F64	VNMLS
FMSCS	VNMLS.F32	
FMSR	VMOV	VMOV (between general-purpose register and single-precision)
FMSRR	VMOV	VMOV (between two general-purpose registers and two single-precision registers)
FMSTAT	VMRS APSR_nzcv, FPSCR	VMRS
FMULD	VMUL.F64	VMUL (floating-point)
FMULS	VMUL.F32	
FMXR	VMSR	VMSR
FNEGD	VNEG.F64	VNEG
FNEGS	VNEG.F32	
FNMACD	VMLS.F64	VNMLS
FNMACS	VMLS.F32	
FNMSCD	VNMLA.F64	VNMLA
FNMSCS	VNMLA.F32	
FNMULD	VNMUL.F64	VNMUL
FNMULS	VNMUL.F32	
FSHTOD	VCVT.F64.S16	VCVT (between floating-point and fixed-point, floating-point)
FSHTOS	VCVT.F32.S16	
FSITOD	VCVT.F64.S32	VCVT (between floating-point and integer, Advanced SIMD), VCVTR

Pre-UAL syntax	UAL equivalent	See
FSITOS	VCVT.F32.S32	
FSLTOD	VCVT.F64.S32	VCVT (between floating-point and fixed-point, floating-point)
FSLTOS	VCVT.F32.S32	
FSQRTD	VSQRT.F64	VSQRT
FSQRTS	VSQRT.F32	
FSTD	VSTR	VSTR
FSTMD, FSTMIA	VSTM.F64	VSTM, VSTMDB, VSTMIA
FSTMS	VSTM.F32	
FSTS	VSTR	VSTR
FSUBD	VSUB.F64	VSUB (floating-point)
FSUBS	VSUB.F32	
FTOSHD	VCVT.S16.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOSHS	VCVT.S16.F23	
FTOSID	VCVT.S32.F64	VCVT (between floating-point and integer, Advanced SIMD)
FTOSIS	VCVT.S32.F32	
FTOSIZD	VCVTR.S32.F64	VCVTR
FTOSIZS	VCVTR.S32.F32	
FTOSLD	VCVT.S32.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOSLS	VCVT.S32.F32	
FTOUHD	VCVT.U16.F64	
FTOUHS	VCVT.U16.F32	
FTOUID	VCVT.U32.F64	VCVT (between floating-point and integer, Advanced SIMD)
FTOUIS	VCVT.U32.F32	
FTOUIZD	VCVTR.U32.F64	VCVTR
FTOUIZS	VCVTR.U32.F32	
FTOULD	VCVT.U32.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOULS	VCVT.U32.F32	
FUHTOD,	VCVT.F64.U16	
FUHTOS	VCVT.F64.U16	
FUITOD	VCVT.F64.U32	VCVT (between floating-point and integer, Advanced SIMD)
FUITOS	VCVT.F32.U32	
FULTOD	VCVT.F64.U32	VCVT (between floating-point and fixed-point, floating-point)
FULTOS	VCVT.F32.U32	

H3.1.3 FCONST

The syntax of FCONST is

FCONST<dest>{<c>} <Fd>, #<imm8>

where:

<dest> Specifies the destination data type. It must be one of:

S	Single-precision floating-point.				
D	Double-precision floating-point.				
<c>	This is an optional field. It specifies the condition under which the instruction is executed. See Conditional execution for the range of available conditions and their encoding. If <c> is omitted, it defaults to <i>always</i> (AL).				
<Fd>	Specifies the destination register. It must be one of: <table> <tr> <td><Dd></td><td>64-bit name of the SIMD&FP destination register.</td></tr> <tr> <td><Sd></td><td>32-bit name of the SMID&FP destination register.</td></tr> </table>	<Dd>	64-bit name of the SIMD&FP destination register.	<Sd>	32-bit name of the SMID&FP destination register.
<Dd>	64-bit name of the SIMD&FP destination register.				
<Sd>	32-bit name of the SMID&FP destination register.				
<imm8>	Specifies the immediate value used to generate the floating-point constant.				
FCONSTD{<c>} <Dd>, #<imm8> maps to VMOV.F64 <Dd>, #<fpimm>					
FCONSTS{<c>} <Sd>, #<imm8> maps to VMOV.F32 <Sd>, #<fpimm>					

Appendix H4

Arm Pseudocode Definition

The *Architecture Specification Language* (ASL) is defined in *Arm[®] Architecture Specification Language Reference* (ARM DDI 0626).

This appendix provides further information about the pseudocode used in this manual. It contains the following sections:

- [About the Arm pseudocode](#)
- [Pseudocode for instruction descriptions](#)

H4.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions. [Pseudocode for instruction descriptions](#) gives general information about this instruction pseudocode, including its limitations.

H4.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs.

H4.2 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in [General limitations of Arm pseudocode](#) and [Limitations of the instruction pseudocode](#).

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. [Instruction encoding diagrams and instruction pseudocode](#) gives more information about the pseudocode provided for each instruction.

Appendix H5

Registers Index

This appendix provides indexes to the register descriptions in this manual. It contains the following sections:

- [Alphabetical index of AArch32 registers and System instructions](#)
- [Functional index of AArch32 registers and System instructions](#)
- [Alphabetical index of memory-mapped registers](#)
- [Functional index of memory-mapped registers](#)

H5.1 Alphabetical index of AArch32 registers and System instructions

This section is an index of AArch32 registers and System instructions in alphabetical order.

Table H5-1 Alphabetical index of AArch32 Registers

Register	Description, see
ACTLR	Auxiliary Control Register
ACTLR2	Auxiliary Control Register 2
ADFSR	Auxiliary Data Fault Status Register
AIDR	Auxiliary ID Register
AIFSR	Auxiliary Instruction Fault Status Register
AMAIR0	Auxiliary Memory Attribute Indirection Register 0
AMAIR1	Auxiliary Memory Attribute Indirection Register 1
APSR	Application Program Status Register
ATS12NSOPR	Address Translate Stages 1 and 2 Non-secure Only PL1 Read
ATS12NSOPW	Address Translate Stages 1 and 2 Non-secure Only PL1 Write
ATS12NSOUR	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read
ATS12NSOUW	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write
ATS1CPR	Address Translate Stage 1 Current state PL1 Read
ATS1CPW	Address Translate Stage 1 Current state PL1 Write
ATS1CUR	Address Translate Stage 1 Current state Unprivileged Read
ATS1CUW	Address Translate Stage 1 Current state Unprivileged Write
ATS1HR	Address Translate Stage 1 Hyp mode Read
ATS1HW	Address Translate Stage 1 Hyp mode Write
BPIALL	Branch Predictor Invalidate All
BPIALLIS	Branch Predictor Invalidate All, Inner Shareable
BPIMVA	Branch Predictor Invalidate by VA
CCSIDR	Current Cache Size ID Register
CLIDR	Cache Level ID Register
CNTFRQ	Counter-timer Frequency register
CNTHCTL	Counter-timer Hyp Control register
CNTHP_CTL	Counter-timer Hyp Physical Timer Control register
CNTHP_CVAL	Counter-timer Hyp Physical CompareValue register
CNTHP_TVAL	Counter-timer Hyp Physical Timer TimerValue register
CNTKCTL	Counter-timer Kernel Control register
CNTP_CTL	Counter-timer Physical Timer Control register
CNTP_CVAL	Counter-timer Physical Timer CompareValue register
CNTP_TVAL	Counter-timer Physical Timer TimerValue register
CNTPCT	Counter-timer Physical Count register

Register	Description, see
CNTV_CTL	Counter-timer Virtual Timer Control register
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register
CNTVCT	Counter-timer Virtual Count register
CNTVOFF	Counter-timer Virtual Offset register
CONTEXTIDR	Context ID Register
CP15DMB	Data Memory Barrier System instruction
CP15DSB	Data Synchronization Barrier System instruction
CP15ISB	Instruction Synchronization Barrier System instruction
CPACR	Architectural Feature Access Control Register
CPSR	Current Program Status Register
CSSELR	Cache Size Selection Register
CTR	Cache Type Register
DBGAUTHSTATUS	Debug Authentication Status register
DBGCLAIMCLR	Debug CLAIM Tag Clear register
DBGCLAIMSET	Debug CLAIM Tag Set register
DBGDCCINT	DCC Interrupt Enable Register
DBGDEVID	Debug Device ID register 0
DBGDEVID1	Debug Device ID register 1
DBGDEVID2	Debug Device ID register 2
DBGDIDR	Debug ID Register
DBGDRAR	Debug ROM Address Register
DBGDSAR	Debug Self Address Register
DBGDSCRext	Debug Status and Control Register, External View
DBGDSCRint	Debug Status and Control Register, Internal View
DBGDTRRXext	Debug OS Lock Data Transfer Register, Receive, External View
DBGDTRRXint	Debug Data Transfer Register, Receive
DBGDTRTXext	Debug OS Lock Data Transfer Register, Transmit
DBGDTRTXint	Debug Data Transfer Register, Transmit
DBGOSDLR	Debug OS Double Lock Register
DBGOSECCR	Debug OS Lock Exception Catch Control Register
DBGOSLAR	Debug OS Lock Access Register
DBGOSLSR	Debug OS Lock Status Register
DBGPRCR	Debug Power Control Register
DBGVCR	Debug Vector Catch Register
DBGWFAR	Debug Watchpoint Fault Address Register
DCCIMVAC	Data Cache line Clean and Invalidate by VA to PoC
DCCISW	Data Cache line Clean and Invalidate by Set-Way
DCCMVAC	Data Cache line Clean by VA to PoC
DCCMVAU	Data Cache line Clean by VA to PoU

Register	Description, see
DCCSW	Data Cache line Clean by Set-Way
DCIMVAC	Data Cache line Invalidate by VA to PoC
DCISW	Data Cache line Invalidate by Set-Way
DFAR	Data Fault Address Register
DFSR	Data Fault Status Register
DLR	Debug Link Register
DSPSR	Debug Saved Program Status Register
DTLBIALL	Data TLB Invalidate All
DTLBIASID	Data TLB Invalidate by ASID match
DTLBIMVA	Data TLB Invalidate by VA
ELR_hyp	Exception Link Register (Hyp mode)
FCSEIDR	FCSE Process ID register
FPEXC	Floating-Point Exception Control register
FPSCR	Floating-Point Status and Control Register
FPSID	Floating-Point System ID register
HACR	Hyp Auxiliary Configuration Register
HACTLR	Hyp Auxiliary Control Register
HACTLR2	Hyp Auxiliary Control Register 2
HADFSR	Hyp Auxiliary Data Fault Status Register
HAIFSR	Hyp Auxiliary Instruction Fault Status Register
HAMAIR0	Hyp Auxiliary Memory Attribute Indirection Register 0
HAMAIR1	Hyp Auxiliary Memory Attribute Indirection Register 1
HCPTR	Hyp Architectural Feature Trap Register
HCR	Hyp Configuration Register
HCR2	Hyp Configuration Register 2
HDCCR	Hyp Debug Control Register
HDFAR	Hyp Data Fault Address Register
HIFAR	Hyp Instruction Fault Address Register
HMAIR0	Hyp Memory Attribute Indirection Register 0
HMAIR1	Hyp Memory Attribute Indirection Register 1
HMPUIR	Hypervisor MPU Type Register
HPFAR	Hyp IPA Fault Address Register
HPRBAR	Hypervisor Protection Region Base Address Register
HPREN	Hypervisor Protection Region Enable Register
HPRLAR	Hypervisor Protection Region Limit Address Register
HPRSEL	Hypervisor Protection Region Selection Register
HRMR	Hyp Reset Management Register
HSCTLR	Hyp System Control Register
HSR	Hyp Syndrome Register
HSTR	Hyp System Trap Register

Register	Description, see
HTPIDR	Hyp Software Thread ID Register
HVBAR	Hyp Vector Base Address Register
ICC_ASGI1R	Interrupt Controller Alias Software Generated Interrupt Group 1 Register
ICC_BPR0	Interrupt Controller Binary Point Register 0
ICC_BPR1	Interrupt Controller Binary Point Register 1
ICC_CTLR	Interrupt Controller Control Register
ICC_DIR	Interrupt Controller Deactivate Interrupt Register
ICC_EOIR0	Interrupt Controller End Of Interrupt Register 0
ICC_EOIR1	Interrupt Controller End Of Interrupt Register 1
ICC_HPIR0	Interrupt Controller Highest Priority Pending Interrupt Register 0
ICC_HPIR1	Interrupt Controller Highest Priority Pending Interrupt Register 1
ICC_HSRE	Interrupt Controller Hyp System Register Enable register
ICC_IAR0	Interrupt Controller Interrupt Acknowledge Register 0
ICC_IAR1	Interrupt Controller Interrupt Acknowledge Register 1
ICC_IGRPEN0	Interrupt Controller Interrupt Group 0 Enable register
ICC_IGRPEN1	Interrupt Controller Interrupt Group 1 Enable register
ICC_PMR	Interrupt Controller Interrupt Priority Mask Register
ICC_RPR	Interrupt Controller Running Priority Register
ICC_SGI0R	Interrupt Controller Software Generated Interrupt Group 0 Register
ICC_SGI1R	Interrupt Controller Software Generated Interrupt Group 1 Register
ICC_SRE	Interrupt Controller System Register Enable register
ICH_EISR	Interrupt Controller End of Interrupt Status Register
ICH_ELRSR	Interrupt Controller Empty List Register Status Register
ICH_HCR	Interrupt Controller Hyp Control Register
ICH_MISR	Interrupt Controller Maintenance Interrupt State Register
ICH_VMCR	Interrupt Controller Virtual Machine Control Register
ICH_VTR	Interrupt Controller VGIC Type Register
ICIALLU	Instruction Cache Invalidate All to PoU
ICIALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
ICIMVAU	Instruction Cache line Invalidate by VA to PoU
ICV_BPR0	Interrupt Controller Virtual Binary Point Register 0
ICV_BPR1	Interrupt Controller Virtual Binary Point Register 1
ICV_CTLR	Interrupt Controller Virtual Control Register
ICV_DIR	Interrupt Controller Deactivate Virtual Interrupt Register
ICV_EOIR0	Interrupt Controller Virtual End Of Interrupt Register 0
ICV_EOIR1	Interrupt Controller Virtual End Of Interrupt Register 1
ICV_HPIR0	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 0
ICV_HPIR1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 1

Register	Description, see
ICV_IAR0	Interrupt Controller Virtual Interrupt Acknowledge Register 0
ICV_IAR1	Interrupt Controller Virtual Interrupt Acknowledge Register 1
ICV_IGRPEN0	Interrupt Controller Virtual Interrupt Group 0 Enable register
ICV_IGRPEN1	Interrupt Controller Virtual Interrupt Group 1 Enable register
ICV_PMR	Interrupt Controller Virtual Interrupt Priority Mask Register
ICV_RPR	Interrupt Controller Virtual Running Priority Register
ID_AFR0	Auxiliary Feature Register 0
ID_DFR0	Debug Feature Register 0
ID_ISAR0	Instruction Set Attribute Register 0
ID_ISAR1	Instruction Set Attribute Register 1
ID_ISAR2	Instruction Set Attribute Register 2
ID_ISAR3	Instruction Set Attribute Register 3
ID_ISAR4	Instruction Set Attribute Register 4
ID_ISAR5	Instruction Set Attribute Register 5
ID_MMFR0	Memory Model Feature Register 0
ID_MMFR1	Memory Model Feature Register 1
ID_MMFR2	Memory Model Feature Register 2
ID_MMFR3	Memory Model Feature Register 3
ID_MMFR4	Memory Model Feature Register 4
ID_PFR0	Processor Feature Register 0
ID_PFR1	Processor Feature Register 1
IFAR	Instruction Fault Address Register
IFSR	Instruction Fault Status Register
ISR	Interrupt Status Register
ITLBIALL	Instruction TLB Invalidate All
ITLBIASID	Instruction TLB Invalidate by ASID match
ITLBIMVA	Instruction TLB Invalidate by VA
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register
MAIR0	Memory Attribute Indirection Register 0
MAIR1	Memory Attribute Indirection Register 1
MIDR	Main ID Register
MPIDR	Multiprocessor Affinity Register
MPUIR	MPU Type Register (EL1)
MVFR0	Media and VFP Feature Register 0
MVFR1	Media and VFP Feature Register 1
MVFR2	Media and VFP Feature Register 2
NSACR	Non-Secure Access Control Register
PAR	Physical Address Register

Register	Description, see
PMCCFILTR	Performance Monitors Cycle Count Filter Register
PMCCNTR	Performance Monitors Cycle Count Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCNTENCLR	Performance Monitors Count Enable Clear register
PMCNTENSET	Performance Monitors Count Enable Set register
PMCR	Performance Monitors Control Register
PMINTENCLR	Performance Monitors Interrupt Enable Clear register
PMINTENSET	Performance Monitors Interrupt Enable Set register
PMOVSr	Performance Monitors Overflow Flag Status Register
PMOVSSET	Performance Monitors Overflow Flag Status Set register
PMSELR	Performance Monitors Event Counter Selection Register
PMSWINC	Performance Monitors Software Increment register
PMUSERENR	Performance Monitors User Enable Register
PMXVCNTR	Performance Monitors Selected Event Count Register
PMXEVTPER	Performance Monitors Selected Event Type Register
PRBAR	Protection Region Base Address Register (EL1)
PRLAR	Protection Region Limit Address Register (EL1)
PRSELR	Protection Region Selection Register (EL1)
REVIDR	Revision ID Register
RVBAR	Reset Vector Base Address Register
SCTLR	System Control Register
SPSR	Saved Program Status Register
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_hyp	Saved Program Status Register (Hyp mode)
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_svc	Saved Program Status Register (Supervisor mode)
SPSR_und	Saved Program Status Register (Undefined mode)
TCMTR	TCM Type Register
TLBIALL	TLB Invalidate All
TLBIALLH	TLB Invalidate All, Hyp mode
TLBIALLHIS	TLB Invalidate All, Hyp mode, Inner Shareable
TLBIALLIS	TLB Invalidate All, Inner Shareable
TLBIALLNSNH	TLB Invalidate All, Non-Secure Non-Hyp
TLBIALLNSNHIS	TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable
TLBIASID	TLB Invalidate by ASID match
TLBIASIDIS	TLB Invalidate by ASID match, Inner Shareable
TLBIIPAS2	TLB Invalidate by Intermediate Physical Address, Stage 2

Register	Description, see
TLBIIPAS2IS	TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable
TLBIIPAS2L	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level
TLBIIPAS2LIS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable
TLBIMVA	TLB Invalidate by VA
TLBIMVAA	TLB Invalidate by VA, All ASID
TLBIMVAAIS	TLB Invalidate by VA, All ASID, Inner Shareable
TLBIMVAAL	TLB Invalidate by VA, All ASID, Last level
TLBIMVAALIS	TLB Invalidate by VA, All ASID, Last level, Inner Shareable
TLBIMVAH	TLB Invalidate by VA, Hyp mode
TLBIMVAHIS	TLB Invalidate by VA, Hyp mode, Inner Shareable
TLBIMVAIS	TLB Invalidate by VA, Inner Shareable
TLBIMVAL	TLB Invalidate by VA, Last level
TLBIMVALH	TLB Invalidate by VA, Last level, Hyp mode
TLBIMVALHIS	TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable
TLBIMVALIS	TLB Invalidate by VA, Last level, Inner Shareable
TLBTR	TLB Type Register
TPIDRPRW	PL1 Software Thread ID Register
TPIDRURO	PL0 Read-Only Software Thread ID Register
TPIDRURW	PL0 Read-Write Software Thread ID Register
VBAR	Vector Base Address Register
VMPIDR	Virtualization Multiprocessor ID Register
VPIDR	Virtualization Processor ID Register
VSCTLR	Virtualization System Control Register

H5.2 Functional index of AArch32 registers and System instructions

This section is an index of the AArch32 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- [Special-purpose registers.](#)
- [ID registers.](#)
- [Performance monitors System registers.](#)
- [Debug System registers.](#)
- [Generic timer registers.](#)
- [Cache maintenance System instructions.](#)
- [Address translation System instructions.](#)
- [TLB maintenance System instructions.](#)
- [Legacy feature registers and System instructions.](#)
- [Base System registers.](#)

H5.2.1 Special-purpose registers

This section is an index to the registers in the Processor state registers functional group.

Table H5-2 Special-purpose registers

Register	Description, see
APSR	Application Program Status Register
CPSR	Current Program Status Register
DLR	Debug Link Register
DSPSR	Debug Saved Program Status Register
ELR_hyp	Exception Link Register (Hyp mode)
SPSR	Saved Program Status Register
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_hyp	Saved Program Status Register (Hyp mode)
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_svc	Saved Program Status Register (Supervisor mode)
SPSR_und	Saved Program Status Register (Undefined mode)

H5.2.2 ID registers

This section is an index to the registers in the Identification registers functional group.

Table H5-3 ID registers

Register	Description, see
CCSIDR	Current Cache Size ID Register
CLIDR	Cache Level ID Register
CSSELR	Cache Size Selection Register
CTR	Cache Type Register
FPSID	Floating-Point System ID register
ID_AFR0	Auxiliary Feature Register 0
ID_DFR0	Debug Feature Register 0
ID_ISAR0	Instruction Set Attribute Register 0
ID_ISAR1	Instruction Set Attribute Register 1
ID_ISAR2	Instruction Set Attribute Register 2
ID_ISAR3	Instruction Set Attribute Register 3
ID_ISAR4	Instruction Set Attribute Register 4
ID_ISAR5	Instruction Set Attribute Register 5
ID_MMFR0	Memory Model Feature Register 0
ID_MMFR1	Memory Model Feature Register 1
ID_MMFR2	Memory Model Feature Register 2
ID_MMFR3	Memory Model Feature Register 3
ID_MMFR4	Memory Model Feature Register 4
ID_PFR0	Processor Feature Register 0
ID_PFR1	Processor Feature Register 1
MIDR	Main ID Register
MPIDR	Multiprocessor Affinity Register
MVFR0	Media and VFP Feature Register 0
MVFR1	Media and VFP Feature Register 1
MVFR2	Media and VFP Feature Register 2
REVIDR	Revision ID Register
TCMTR	TCM Type Register
TLBTR	TLB Type Register
VMPIDR	Virtualization Multiprocessor ID Register
VPIDR	Virtualization Processor ID Register

H5.2.3 Performance monitors System registers

This section is an index to the System registers in the Performance Monitors registers functional group.

Table H5-4 Performance monitors registers

Register	Description, see
PMCCFILTR	Performance Monitors Cycle Count Filter Register
PMCCNTR	Performance Monitors Cycle Count Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCNTENCLR	Performance Monitors Count Enable Clear register
PMCNTENSET	Performance Monitors Count Enable Set register
PMCR	Performance Monitors Control Register
PMINTENCLR	Performance Monitors Interrupt Enable Clear register
PMINTENSET	Performance Monitors Interrupt Enable Set register
PMOVSr	Performance Monitors Overflow Flag Status Register
PMOVSSET	Performance Monitors Overflow Flag Status Set register
PMSCLR	Performance Monitors Event Counter Selection Register
PMSWINC	Performance Monitors Software Increment register
PMUSERENR	Performance Monitors User Enable Register
PMXEVCNTR	Performance Monitors Selected Event Count Register
PMXEVTPER	Performance Monitors Selected Event Type Register

H5.2.4 Debug System registers

This section is an index to the System registers in the Debug registers functional group.

Table H5-5 Debug registers

Register	Description, see
DBGAUTHSTATUS	Debug Authentication Status register
DBGCLAIMCLR	Debug CLAIM Tag Clear register
DBGCLAIMSET	Debug CLAIM Tag Set register
DBGDCCINT	DCC Interrupt Enable Register
DBGDEVID	Debug Device ID register 0
DBGDEVID1	Debug Device ID register 1
DBGDEVID2	Debug Device ID register 2
DBGDIDR	Debug ID Register
DBGDRAR	Debug ROM Address Register
DBGDSAR	Debug Self Address Register
DBGDSCRExt	Debug Status and Control Register, External View
DBGDSCRInt	Debug Status and Control Register, Internal View
DBGDTRRXext	Debug OS Lock Data Transfer Register, Receive, External View

Register	Description, see
DBGDTRRXint	Debug Data Transfer Register, Receive
DBGDTRTXext	Debug OS Lock Data Transfer Register, Transmit
DBGDTRTXint	Debug Data Transfer Register, Transmit
DBGOSDLR	Debug OS Double Lock Register
DBGOSECCR	Debug OS Lock Exception Catch Control Register
DBGOSLAR	Debug OS Lock Access Register
DBGOSLSR	Debug OS Lock Status Register
DBGPRCR	Debug Power Control Register
DBGVCR	Debug Vector Catch Register
DBGWFAR	Debug Watchpoint Fault Address Register
HDCR	Hyp Debug Control Register

H5.2.5 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table H5-6 Generic timer registers

Register	Description, see
CNTRQ	Counter-timer Frequency register
CNTHCTL	Counter-timer Hyp Control register
CNTHP_CTL	Counter-timer Hyp Physical Timer Control register
CNTHP_CVAL	Counter-timer Hyp Physical CompareValue register
CNTHP_TVAL	Counter-timer Hyp Physical Timer TimerValue register
CNTKCTL	Counter-timer Kernel Control register
CNTP_CTL	Counter-timer Physical Timer Control register
CNTP_CVAL	Counter-timer Physical Timer CompareValue register
CNTP_TVAL	Counter-timer Physical Timer TimerValue register
CNTPCT	Counter-timer Physical Count register
CNTV_CTL	Counter-timer Virtual Timer Control register
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register
CNTVCT	Counter-timer Virtual Count register
CNTVOFF	Counter-timer Virtual Offset register

H5.2.6 Cache maintenance System instructions

This section is an index to the System instructions in the Cache maintenance instructions functional group.

Table H5-7 Cache maintenance system instructions

System instruction	Description, see
BPIALL	Branch Predictor Invalidate All
BPIALLIS	Branch Predictor Invalidate All, Inner Shareable
BPIMVA	Branch Predictor Invalidate by VA
DCCIMVAC	Data Cache line Clean and Invalidate by VA to PoC
DCCISW	Data Cache line Clean and Invalidate by Set-Way
DCCMVAC	Data Cache line Clean by VA to PoC
DCCMVAU	Data Cache line Clean by VA to PoU
DCCSW	Data Cache line Clean by Set-Way
DCIMVAC	Data Cache line Invalidate by VA to PoC
DCISW	Data Cache line Invalidate by Set-Way
ICIALLU	Instruction Cache Invalidate All to PoU
ICIALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
ICIMVAU	Instruction Cache line Invalidate by VA to PoU

H5.2.7 Address translation System instructions

This section is an index to the System instructions in the Address translation instructions functional group.

Table H5-8 Address translation system instructions

System instruction	Description, see
ATS12NSOPR	Address Translate Stages 1 and 2 Non-secure Only PL1 Read
ATS12NSOPW	Address Translate Stages 1 and 2 Non-secure Only PL1 Write
ATS12NSOUR	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read
ATS12NSOUW	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write
ATS1CPR	Address Translate Stage 1 Current state PL1 Read
ATS1CPW	Address Translate Stage 1 Current state PL1 Write
ATS1CUR	Address Translate Stage 1 Current state Unprivileged Read
ATS1CUW	Address Translate Stage 1 Current state Unprivileged Write
ATS1HR	Address Translate Stage 1 Hyp mode Read
ATS1HW	Address Translate Stage 1 Hyp mode Write

H5.2.8 TLB maintenance System instructions

This section is an index to the System instructions in the TLB maintenance instructions functional group.

Table H5-9 TLB maintenance system instructions

System instruction	Description, see
DTLBIALL	Data TLB Invalidate All
DTLBIASID	Data TLB Invalidate by ASID match
DTLBIMVA	Data TLB Invalidate by VA
ITLBIALL	Instruction TLB Invalidate All
ITLBIASID	Instruction TLB Invalidate by ASID match
ITLBIMVA	Instruction TLB Invalidate by VA
TLBIALL	TLB Invalidate All
TLBIALLH	TLB Invalidate All, Hyp mode
TLBIALLHIS	TLB Invalidate All, Hyp mode, Inner Shareable
TLBIALLIS	TLB Invalidate All, Inner Shareable
TLBIALLNSNH	TLB Invalidate All, Non-Secure Non-Hyp
TLBIALLNSNHIS	TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable
TLBIASID	TLB Invalidate by ASID match
TLBIASIDIS	TLB Invalidate by ASID match, Inner Shareable
TLBIIPAS2	TLB Invalidate by Intermediate Physical Address, Stage 2
TLBIIPAS2IS	TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable
TLBIIPAS2L	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level
TLBIIPAS2LIS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable
TLBIMVA	TLB Invalidate by VA
TLBIMVAA	TLB Invalidate by VA, All ASID
TLBIMVAAIS	TLB Invalidate by VA, All ASID, Inner Shareable
TLBIMVAAL	TLB Invalidate by VA, All ASID, Last level
TLBIMVAALIS	TLB Invalidate by VA, All ASID, Last level, Inner Shareable
TLBIMVAH	TLB Invalidate by VA, Hyp mode
TLBIMVAHIS	TLB Invalidate by VA, Hyp mode, Inner Shareable
TLBIMVAIS	TLB Invalidate by VA, Inner Shareable
TLBIMVAL	TLB Invalidate by VA, Last level
TLBIMVALH	TLB Invalidate by VA, Last level, Hyp mode
TLBIMVALHIS	TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable
TLBIMVALIS	TLB Invalidate by VA, Last level, Inner Shareable

H5.2.9 Prediction restriction instructions

This section is an index to the registers in the Prediction restriction instructions functional group.

Table H5-10 Cache maintenance system instructions

Description, see

H5.2.10 Legacy feature registers and system instructions

This section is an index to the registers in the Legacy feature registers functional group.

Table H5-11 Legacy feature registers and system instructions

Register	Description, see
CP15DMB	Data Memory Barrier System instruction
CP15DSB	Data Synchronization Barrier System instruction
CP15ISB	Instruction Synchronization Barrier System instruction
FCSEIDR	FCSE Process ID register
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register

H5.2.11 Base System registers

This section is an index to the System registers in the Base System registers functional group.

Table H5-12 Base System registers

Register	Description, see
CPACR	Architectural Feature Access Control Register
FCSEIDR	FCSE Process ID register
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register
SCTLR	System Control Register

H5.3 Alphabetical index of memory-mapped registers

This section is an index of memory-mapped registers in alphabetical order.

Table H5-13 Alphabetical index of Memory-Mapped Registers

Register	Description, see
ASICCTL	CTI External Multiplexer Control register
CNTCR	Counter Control Register
CNTCV	Counter Count Value register
CNTEL0ACR	Counter-timer EL0 Access Control Register
CNTFID0	Counter Frequency ID
CNTFRQ	Counter-timer Frequency
CNTNSAR	Counter-timer Non-secure Access Register
CNTP_CTL	Counter-timer Physical Timer Control
CNTP_CVAL	Counter-timer Physical Timer Compare Value
CNTP_TVAL	Counter-timer Physical Timer Timer Value
CNTPCT	Counter-timer Physical Count
CNTSR	Counter Status Register
CNTTIDR	Counter-timer Timer ID Register
CNTV_CTL	Counter-timer Virtual Timer Control
CNTV_CVAL	Counter-timer Virtual Timer Compare Value
CNTV_TVAL	Counter-timer Virtual Timer Timer Value
CNTVCT	Counter-timer Virtual Count
CNTVOFF	Counter-timer Virtual Offset
CTIAPPCLEAR	CTI Application Trigger Clear register
CTIAPPULSE	CTI Application Pulse register
CTIAPPSET	CTI Application Trigger Set register
CTIAUTHSTATUS	CTI Authentication Status register
CTICHINSTATUS	CTI Channel In Status register
CTICHOUTSTATUS	CTI Channel Out Status register
CTICIDR0	CTI Component Identification Register 0
CTICIDR1	CTI Component Identification Register 1
CTICIDR2	CTI Component Identification Register 2
CTICIDR3	CTI Component Identification Register 3
CTICLAIMCLR	CTI CLAIM Tag Clear register
CTICLAIMSET	CTI CLAIM Tag Set register
CTICONTROL	CTI Control register
CTIDEVAFF0	CTI Device Affinity register 0
CTIDEVAFF1	CTI Device Affinity register 1
CTIDEVARCH	CTI Device Architecture register

Register	Description, see
CTIDEVID	CTI Device ID register 0
CTIDEVID1	CTI Device ID register 1
CTIDEVID2	CTI Device ID register 2
CTIDEVTYPE	CTI Device Type register
CTIGATE	CTI Channel Gate Enable register
CTIINTACK	CTI Output Trigger Acknowledge register
CTIITCTRL	CTI Integration mode Control register
CTILAR	CTI Lock Access Register
CTILSR	CTI Lock Status Register
CTIPIDR0	CTI Peripheral Identification Register 0
CTIPIDR1	CTI Peripheral Identification Register 1
CTIPIDR2	CTI Peripheral Identification Register 2
CTIPIDR3	CTI Peripheral Identification Register 3
CTIPIDR4	CTI Peripheral Identification Register 4
CTIPIDR5	CTI Peripheral Identification Register 5
CTIPIDR6	CTI Peripheral Identification Register 6
CTIPIDR7	CTI Peripheral Identification Register 7
CTITRIGINSTATUS	CTI Trigger In Status register
CTITRIGOUTSTATUS	CTI Trigger Out Status register
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDACR	External Debug Auxiliary Control Register
EDCIDR0	External Debug Component Identification Register 0
EDCIDR1	External Debug Component Identification Register 1
EDCIDR2	External Debug Component Identification Register 2
EDCIDR3	External Debug Component Identification Register 3
EDCIDSR	External Debug Context ID Sample Register
EDDEVAFF0	External Debug Device Affinity register 0
EDDEVAFF1	External Debug Device Affinity register 1
EDDEVARCH	External Debug Device Architecture Register
EDDEVID	External Debug Device ID register 0
EDDEVID1	External Debug Device ID Register 1
EDDEVID2	External Debug Device ID register 2
EDDEVTYPE	External Debug Device Type register
EDDFR	External Debug Feature Register
EDECCR	External Debug Exception Catch Control Register

Register	Description, see
EDECR	External Debug Execution Control Register
EDESR	External Debug Event Status Register
EDITCTRL	External Debug Integration mode Control register
EDITR	External Debug Instruction Transfer Register
EDLAR	External Debug Lock Access Register
EDLSR	External Debug Lock Status Register
EDPCSR	External Debug Program Counter Sample Register
EDPFR	External Debug Processor Feature Register
EDPIDR0	External Debug Peripheral Identification Register 0
EDPIDR1	External Debug Peripheral Identification Register 1
EDPIDR2	External Debug Peripheral Identification Register 2
EDPIDR3	External Debug Peripheral Identification Register 3
EDPIDR4	External Debug Peripheral Identification Register 4
EDPIDR5	External Debug Peripheral Identification Register 5
EDPIDR6	External Debug Peripheral Identification Register 6
EDPIDR7	External Debug Peripheral Identification Register 7
EDPRCR	External Debug Power-Reset Control Register
EDPRSR	External Debug Processor Status Register
EDRCR	External Debug Reserve Control Register
EDSCR	External Debug Status and Control Register
EDVIDSR	External Debug Virtual Context Sample Register
EDWAR	External Debug Watchpoint Address Register
ERRCIDR0	Component Identification Register 0
ERRCIDR1	Component Identification Register 1
ERRCIDR2	Component Identification Register 2
ERRCIDR3	Component Identification Register 3
ERRDEVAFF	Device Affinity Register
ERRDEVARCH	Device Architecture Register
ERRDEVID	Device Configuration Register
ERRPIDR0	Peripheral Identification Register 0
ERRPIDR1	Peripheral Identification Register 1
ERRPIDR2	Peripheral Identification Register 2
ERRPIDR3	Peripheral Identification Register 3
ERRPIDR4	Peripheral Identification Register 4
GICC_ABPR	CPU Interface Aliased Binary Point Register
GICC_AEOIR	CPU Interface Aliased End Of Interrupt Register
GICC_AHPPIR	CPU Interface Aliased Highest Priority Pending Interrupt Register
GICC_AIAR	CPU Interface Aliased Interrupt Acknowledge Register
GICC_BPR	CPU Interface Binary Point Register

Register	Description, see
GICC_CTLR	CPU Interface Control Register
GICC_DIR	CPU Interface Deactivate Interrupt Register
GICC_EOIR	CPU Interface End Of Interrupt Register
GICC_HPIR	CPU Interface Highest Priority Pending Interrupt Register
GICC_IAR	CPU Interface Interrupt Acknowledge Register
GICC_IIDR	CPU Interface Identification Register
GICC_PMR	CPU Interface Priority Mask Register
GICC_RPR	CPU Interface Running Priority Register
GICC_STATUSR	CPU Interface Status Register
GICD_CLRSPI_NSR	Clear Non-secure SPI Pending Register
GICD_CLRSPI_SR	Clear Secure SPI Pending Register
GICD_CTLR	Distributor Control Register
GICD_IIDR	Distributor Implementer Identification Register
GICD_SETSPI_NSR	Set Non-secure SPI Pending Register
GICD_SETSPI_SR	Set Secure SPI Pending Register
GICD_SGIR	Software Generated Interrupt Register
GICD_STATUSR	Error Reporting Status Register
GICD_TYPER	Interrupt Controller Type Register
GICD_TYPER2	Interrupt Controller Type Register 2
GICH_EISR	End Interrupt Status Register
GICH_ELRSR	Empty List Register Status Register
GICH_HCR	Hypervisor Control Register
GICH_MISR	Maintenance Interrupt Status Register
GICH_VMCR	Virtual Machine Control Register
GICH_VTR	Virtual Type Register
GICM_CLRSPI_NSR	Clear Non-secure SPI Pending Register
GICM_CLRSPI_SR	Clear Secure SPI Pending Register
GICM_IIDR	Distributor Implementer Identification Register
GICM_SETSPI_NSR	Set Non-secure SPI Pending Register
GICM_SETSPI_SR	Set Secure SPI Pending Register
GICM_TYPER	Distributor MSI Type Register
GICR_CLRLPIR	Clear LPI Pending Register
GICR_CTLR	Redistributor Control Register
GICR_ICACTIVER0	Interrupt Clear-Active Register 0
GICR_ICENABLER0	Interrupt Clear-Enable Register 0
GICR_ICFGR0	Interrupt Configuration Register 0
GICR_ICFGR1	Interrupt Configuration Register 1
GICR_ICPENDR0	Interrupt Clear-Pending Register 0
GICR_IGROUPR0	Interrupt Group Register 0
GICR_IGRPMODR0	Interrupt Group Modifier Register 0

Register	Description, see
GICR_IIDR	Redistributor Implementer Identification Register
GICR_INVALLR	Redistributor Invalidate All Register
GICR_INVLPIR	Redistributor Invalidate LPI Register
GICR_ISACTIVER0	Interrupt Set-Active Register 0
GICR_ISENBALER0	Interrupt Set-Enable Register 0
GICR_ISPENDR0	Interrupt Set-Pending Register 0
GICR_MPAMIDR	Report maximum PARTID and PMG Register
GICR_NSACR	Non-secure Access Control Register
GICR_PARTIDR	Set PARTID and PMG Register
GICR_PENDBASER	Redistributor LPI Pending Table Base Address Register
GICR_PROPBASER	Redistributor Properties Base Address Register
GICR_SETLPIR	Set LPI Pending Register
GICR_STATUSR	Error Reporting Status Register
GICR_SYNCR	Redistributor Synchronize Register
GICR_TYPER	Redistributor Type Register
GICR_VPENDBASER	Virtual Redistributor LPI Pending Table Base Address Register
GICR_VPROPBASER	Virtual Redistributor Properties Base Address Register
GICR_WAKER	Redistributor Wake Register
GICV_ABPR	Virtual Machine Aliased Binary Point Register
GICV_AEOIR	Virtual Machine Aliased End Of Interrupt Register
GICV_AHPPIR	Virtual Machine Aliased Highest Priority Pending Interrupt Register
GICV_AIAR	Virtual Machine Aliased Interrupt Acknowledge Register
GICV_BPR	Virtual Machine Binary Point Register
GICV_CTLR	Virtual Machine Control Register
GICV_DIR	Virtual Machine Deactivate Interrupt Register
GICV_EOIR	Virtual Machine End Of Interrupt Register
GICV_HPPIR	Virtual Machine Highest Priority Pending Interrupt Register
GICV_IAR	Virtual Machine Interrupt Acknowledge Register
GICV_IIDR	Virtual Machine CPU Interface Identification Register
GICV_PMR	Virtual Machine Priority Mask Register
GICV_RPR	Virtual Machine Running Priority Register
GICV_STATUSR	Virtual Machine Error Reporting Status Register
GITS_CBASER	ITS Command Queue Descriptor
GITS_CREADR	ITS Read Register
GITS_CTLR	ITS Control Register
GITS_CWRITER	ITS Write Register
GITS_IIDR	ITS Identification Register
GITS_MPAMIDR	Report maximum PARTID and PMG Register

Register	Description, see
GITS_PARTIDR	Set PARTID and PMG Register
GITS_STATUSR	ITS Error Reporting Status Register
GITS_TRANSLATER	ITS Translation Register
GITS_TYPER	ITS Type Register
GITS_UMSIR	ITS Unmapped MSI register
MIDR_EL1	Main ID Register
OSLAR_EL1	OS Lock Access Register
PMAUTHSTATUS	Performance Monitors Authentication Status register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Counter
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCFGR	Performance Monitors Configuration Register
PMCIDR0	Performance Monitors Component Identification Register 0
PMCIDR1	Performance Monitors Component Identification Register 1
PMCIDR2	Performance Monitors Component Identification Register 2
PMCIDR3	Performance Monitors Component Identification Register 3
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMDEVAFF0	Performance Monitors Device Affinity register 0
PMDEVAFF1	Performance Monitors Device Affinity register 1
PMDEVARCH	Performance Monitors Device Architecture register
PMDEVTYPE	Performance Monitors Device Type register
PMIIDR	Performance Monitors Implementation Identification Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMITCTRL	Performance Monitors Integration mode Control register
PMLAR	Performance Monitors Lock Access Register
PMLSR	Performance Monitors Lock Status Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMPIDR0	Performance Monitors Peripheral Identification Register 0
PMPIDR1	Performance Monitors Peripheral Identification Register 1
PMPIDR2	Performance Monitors Peripheral Identification Register 2

Register	Description, see
PMPIDR3	Performance Monitors Peripheral Identification Register 3
PMPIDR4	Performance Monitors Peripheral Identification Register 4
PMPIDR5	Performance Monitors Peripheral Identification Register 5
PMPIDR6	Performance Monitors Peripheral Identification Register 6
PMPIDR7	Performance Monitors Peripheral Identification Register 7
PMSWINC_EL0	Performance Monitors Software Increment Register

H5.4 Functional index of memory-mapped registers

This section is an index of the memory-mapped registers, divided by functional group. Each of the following sections lists the registers for a functional group:

- [ID registers.](#)
- [Memory-mapped Performance monitors registers.](#)
- [Generic timer registers.](#)
- [Debug registers.](#)
- [Cross-trigger interface registers.](#)

H5.4.1 ID registers

This section is an index to the registers in the Identification registers functional group.

Table H5-14 ID registers

Register	Description, see
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDDFR	External Debug Feature Register
EDPFR	External Debug Processor Feature Register
MIDR_EL1	Main ID Register

H5.4.2 Memory-mapped Performance monitors registers

This section is an index to the memory-mapped registers in the Performance Monitors registers functional group.

Table H5-15 Performance monitors registers

Register	Description, see
PMAUTHSTATUS	Performance Monitors Authentication Status register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Counter
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCFGR	Performance Monitors Configuration Register
PMCIDR0	Performance Monitors Component Identification Register 0
PMCIDR1	Performance Monitors Component Identification Register 1
PMCIDR2	Performance Monitors Component Identification Register 2

Register	Description, see
PMCIDR3	Performance Monitors Component Identification Register 3
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMDEVAFF0	Performance Monitors Device Affinity register 0
PMDEVAFF1	Performance Monitors Device Affinity register 1
PMDEVARCH	Performance Monitors Device Architecture register
PMDEVTYPE	Performance Monitors Device Type register
PMIIDR	Performance Monitors Implementation Identification Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMITCTRL	Performance Monitors Integration mode Control register
PMLAR	Performance Monitors Lock Access Register
PMLSR	Performance Monitors Lock Status Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMPIDR0	Performance Monitors Peripheral Identification Register 0
PMPIDR1	Performance Monitors Peripheral Identification Register 1
PMPIDR2	Performance Monitors Peripheral Identification Register 2
PMPIDR3	Performance Monitors Peripheral Identification Register 3
PMPIDR4	Performance Monitors Peripheral Identification Register 4
PMPIDR5	Performance Monitors Peripheral Identification Register 5
PMPIDR6	Performance Monitors Peripheral Identification Register 6
PMPIDR7	Performance Monitors Peripheral Identification Register 7
PMSWINC_EL0	Performance Monitors Software Increment Register

H5.4.3 Generic timer registers

This section is an index to the registers in the Generic timer registers functional group.

Table H5-16 Generic timer registers

Register	Description, see
CNTCR	Counter Control Register
CNTCV	Counter Count Value register
CNTEL0ACR	Counter-timer EL0 Access Control Register
CNTFID0	Counter Frequency ID
CNTFRQ	Counter-timer Frequency

Register	Description, see
CNTNSAR	Counter-timer Non-secure Access Register
CNTP_CTL	Counter-timer Physical Timer Control
CNTP_CVAL	Counter-timer Physical Timer Compare Value
CNTP_TVAL	Counter-timer Physical Timer Timer Value
CNTPCT	Counter-timer Physical Count
CNTSR	Counter Status Register
CNTTIDR	Counter-timer Timer ID Register
CNTV_CTL	Counter-timer Virtual Timer Control
CNTV_CVAL	Counter-timer Virtual Timer Compare Value
CNTV_TVAL	Counter-timer Virtual Timer Timer Value
CNTVCT	Counter-timer Virtual Count
CNTVOFF	Counter-timer Virtual Offset

H5.4.4 Memory-mapped Debug registers

This section is an index to the memory-mapped registers in the Debug registers functional group.

Table H5-17 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
EDACR	External Debug Auxiliary Control Register
EDCIDR0	External Debug Component Identification Register 0
EDCIDR1	External Debug Component Identification Register 1
EDCIDR2	External Debug Component Identification Register 2
EDCIDR3	External Debug Component Identification Register 3
EDCIDS	External Debug Context ID Sample Register
EDDEVAFF0	External Debug Device Affinity register 0
EDDEVAFF1	External Debug Device Affinity register 1
EDDEVARCH	External Debug Device Architecture Register
EDDEVID	External Debug Device ID register 0
EDDEVID1	External Debug Device ID Register 1

Register	Description, see
EDDEVID2	External Debug Device ID register 2
EDDEVTYPE	External Debug Device Type register
EDECCR	External Debug Exception Catch Control Register
EDECR	External Debug Execution Control Register
EDESR	External Debug Event Status Register
EDITCTRL	External Debug Integration mode Control register
EDITR	External Debug Instruction Transfer Register
EDLAR	External Debug Lock Access Register
EDLSR	External Debug Lock Status Register
EDPCSR	External Debug Program Counter Sample Register
EDPIDR0	External Debug Peripheral Identification Register 0
EDPIDR1	External Debug Peripheral Identification Register 1
EDPIDR2	External Debug Peripheral Identification Register 2
EDPIDR3	External Debug Peripheral Identification Register 3
EDPIDR4	External Debug Peripheral Identification Register 4
EDPIDR5	External Debug Peripheral Identification Register 5
EDPIDR6	External Debug Peripheral Identification Register 6
EDPIDR7	External Debug Peripheral Identification Register 7
EDPRCR	External Debug Power-Reset Control Register
EDPRSR	External Debug Processor Status Register
EDRCR	External Debug Reserve Control Register
EDSCR	External Debug Status and Control Register
EDVIDSR	External Debug Virtual Context Sample Register
EDWAR	External Debug Watchpoint Address Register
OSLAR_EL1	OS Lock Access Register

H5.4.5 Cross-trigger interface registers

This section is an index to the registers in the Cross-Trigger Interface registers functional group.

Table H5-18 Cross-trigger interface registers

Register	Description, see
ASICCTL	CTI External Multiplexer Control register
CTIAPPCLEAR	CTI Application Trigger Clear register
CTIAPPULSE	CTI Application Pulse register
CTIAPPSET	CTI Application Trigger Set register
CTIAUTHSTATUS	CTI Authentication Status register
CTICHINSTATUS	CTI Channel In Status register
CTICHOUTSTATUS	CTI Channel Out Status register
CTICIDR0	CTI Component Identification Register 0
CTICIDR1	CTI Component Identification Register 1
CTICIDR2	CTI Component Identification Register 2
CTICIDR3	CTI Component Identification Register 3
CTICLAIMCLR	CTI CLAIM Tag Clear register
CTICLAIMSET	CTI CLAIM Tag Set register
CTICONTROL	CTI Control register
CTIDEVAFF0	CTI Device Affinity register 0
CTIDEVAFF1	CTI Device Affinity register 1
CTIDEVARCH	CTI Device Architecture register
CTIDEVID	CTI Device ID register 0
CTIDEVID1	CTI Device ID register 1
CTIDEVID2	CTI Device ID register 2
CTIDEVTYPE	CTI Device Type register
CTIGATE	CTI Channel Gate Enable register
CTIINTACK	CTI Output Trigger Acknowledge register
CTIITCTRL	CTI Integration mode Control register
CTILAR	CTI Lock Access Register
CTILSR	CTI Lock Status Register
CTIPIDR0	CTI Peripheral Identification Register 0
CTIPIDR1	CTI Peripheral Identification Register 1
CTIPIDR2	CTI Peripheral Identification Register 2
CTIPIDR3	CTI Peripheral Identification Register 3
CTIPIDR4	CTI Peripheral Identification Register 4
CTIPIDR5	CTI Peripheral Identification Register 5
CTIPIDR6	CTI Peripheral Identification Register 6
CTIPIDR7	CTI Peripheral Identification Register 7
CTITRIGINSTATUS	CTI Trigger In Status register
CTITRIGOUTSTATUS	CTI Trigger Out Status register

Glossary

A32 instruction A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in A32 state. A32 instructions must be word-aligned.

A32 instructions were previously called ARM instructions.

See also [A32 state](#), [A64 instruction](#), [T32 instruction](#).

A32 state The AArch32 Instruction set state in which the PE executes A32 instructions.

A32 state was previously called ARM state.

See also [T32 instruction](#), [T32 state](#).

A64 instruction A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch64. A64 instructions must be word-aligned. See also [A32 instruction](#), [T32 instruction](#).

AArch32 The 32-bit Execution state. In AArch32 state, addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

See also [AArch64](#), [A32 instruction](#), [T32 instruction](#).

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

See also [AArch32](#), [A64 instruction](#).

Abort An exception caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.

Active element An Active element is an SVE vector element or predicate element that is a source register element or destination register element used by an SVE instruction. When the corresponding element in the instruction's Governing predicate is TRUE, the element is Active. If an instruction is unpredicated, all of the vector elements or predicate elements are implicitly Active.

Addressing mode Means a method for generating the memory address used by a load/store instruction.

Advanced SIMD A feature of the Arm architecture that provides SIMD operations on a register file of SIMD and floating-point registers. Where an implementation supports both Advanced SIMD and floating-point instructions, these instructions operate on the same register file.

- Aligned** A data item stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.
- An aligned access is one where the address of the access is aligned to the size of each element of the access.
- AMBA** Advanced Microcontroller Bus Architecture. The AMBA family of protocol specifications is the Arm open standard for on-chip buses. AMBA provides solutions for the interconnection and management of the functional blocks that make up a *System-on-Chip* (SoC). Applications include the development of embedded systems with one or more processors or signal processors and multiple peripherals.
- Analysis of the trace element stream** Refers to the process of:
- Tracing elements that carry information that a trace analyzer requires to enable it to analyze the trace successfully.
 - Tracing elements that either directly indicate program execution, or carry information about program execution.
- A trace element stream might also contain trace elements that contain timing information. This term is distinct from analysis of program execution.
- Architecturally executed** An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired, it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a NOP because it fails its condition code check, is an architecturally executed instruction.
- In a PE that performs speculative execution, an instruction is not architecturally executed if the PE discards the results of a speculative execution.
- See also [Condition code check](#), [Simple sequential execution](#).
- Architecturally mapped** Where this manual describes a register as being *architecturally mapped* to another register, this indicates that, in an implementation that supports both of the registers, the two registers access the same state.
- Architecturally resolved** The architectural state (S) associated with an operation (O) is architecturally resolved when S is determined to be identical to the architectural state of O that is visible to the programmer through the execution of the program.
- Architecturally UNKNOWN** An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of [UNKNOWN](#). Implementations can define the value of the field, but are not required to do so.
- See also [IMPLEMENTATION DEFINED](#).
- Architecturally Allowed Execution** An *Architecturally Allowed Execution* must satisfy the [External visibility requirement](#).
- ARM core registers** Some older documentation uses *ARM core registers* to refer to the following set of registers for execution in AArch32 state:
- The 13 general-purpose registers, R0-R12, that software can use for processing.
 - SP, the *stack pointer*, that can also be referred to as R13.
 - LR, the *link register*, that can also be referred to as R14.
 - PC, the *Program Counter*, that can also be referred to as R15.
- See also [General-purpose registers](#).
- ARM instruction** See [A32 instruction](#).
- Associativity** See [Cache associativity](#).
- Asynchronous accumulation** Faults that are accumulated in a status register, where the update to the register is asynchronous to the instruction that causes the fault.

Atomicity	Describes either single-copy atomicity or multi-copy atomicity. Atomicity in the Arm architecture defines these forms of atomicity for the Arm architecture. See also Multi-copy atomicity and Single-copy atomicity . Might also refer to poison-atomicity. See Poison atomicity .
Availability	Readiness for correct service.
Banked register	A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.
Baseboard Management Controller	A PE dedicated to system control and monitoring.
Base register	A register specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.
Base register writeback	Describes writing back a modified value to the base register used in an address calculation.
Behaves as if	Where this manual indicates that a PE <i>behaves as if</i> a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.
Big-endian memory	Means that, for example: • A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address. • A byte at a halfword-aligned address is the most significant byte in the halfword at that address. See also Endianness , Little-endian memory .
Blocking	Describes an operation that does not permit following instructions to be executed before the operation completes. A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.
Branch History Buffer (BHB)	The Branch History Buffer is an internal structure that stores branch instruction information including the history of previously executed branches.
Branch prediction	Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target. See also Prefetching .
Breakpoint	A debug event triggered by the execution of a particular instruction, specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.
Built-in self test (BIST)	A mechanism that permits a machine to test itself.
Burst	A group of transfers that form a single transaction.
BWA	BandWidth Allocation.
BWPBM	BandWidth Portion Bit Map.
Byte	An 8-bit data item.
CE	See Corrected Error .
Cache associativity	The number of locations in a cache set to which an address can be assigned. Each location is identified by its <i>way</i> value.
Cache level	The position of a cache in the cache hierarchy. In the Arm architecture, the lower numbered levels are those closest to the PE. For more information, see Terms used in describing the cache maintenance instructions.

- Cache line** The basic unit of storage in a cache. Its size in words is always a power of two, usually 4 or 8 words. A cache line must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.
- Cache lockdown** Enables critical software and data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. It alleviates the delays caused by accessing a cache in a worst-case situation. This ensures that all subsequent accesses to the software and data concerned are cache hits and so complete quickly.
- Cache miss** A memory access that cannot be processed at high speed because the data it addresses is not in the cache.
- Cache sets** Areas of a cache, divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.
- Cache way** A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to (Associativity-1). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way *n* consists of the cache line with index *n* from each cache set.
- Coherence order** See [Coherent](#).
- Coherent** Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.
- Commit window** The Commit window defines the range of *P0 elements* that are committed by a *Commit element*. The oldest *P0 element* in the Commit window is the first *P0 element* committed when a *Commit element* occurs. By default, the Commit window starts on the oldest uncommitted *P0 element* and moves forward to the next uncommitted *P0 element*, with each *P0 element* committed by a *Commit element*. The Commit Window Move element moves the start of the Commit window by a number of *P0 elements*, to allow a *Commit element* to commit *P0 elements* that are younger than the oldest uncommitted *P0 elements*, leaving these older *P0 elements* uncommitted.
- Completer** An agent in a computing system that responds to and completes a memory transaction that was initiated by a Requester.
See also [Requester](#).
- Constructive instruction encoding** A constructive instruction encoding is an instruction encoding where the destination register is encoded independently of the source registers.
- Condition code check** The process of determining whether a conditional instruction executes normally or is treated as a *NOP*. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of *PSTATE.IT* determines whether the instruction is executed normally.
See also [Condition code field](#), [Condition flags](#), [Conditional execution](#).
- Condition code field** A 4-bit field in an instruction that specifies the condition under which the instruction executes.
See also [Condition code check](#).
- Condition flags** The N, Z, C, and V bits of *PSTATE*, an *SPSR*, or *FPSCR*. See the register descriptions for more information.
See also [Condition code check](#), [PSTATE](#).
- Conditional execution** When a conditional instruction starts executing, if the condition code check returns *TRUE*, the instruction executes normally. Otherwise, it is treated as a *NOP*.
See also [Condition code check](#).
- CONSTRAINED UNPREDICTABLE** Where an instruction can result in *UNPREDICTABLE* behavior, the Armv8 and later architectures specify a narrow range of permitted behaviors. This range is the range of *CONSTRAINED UNPREDICTABLE* behavior. All implementations that are compliant with the architecture must follow the *CONSTRAINED UNPREDICTABLE* behavior.

Execution at Non-secure EL1 or EL0 of an instruction that is **CONSTRAINED UNPREDICTABLE** can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not **UNPREDICTABLE** and is not **CONSTRAINED UNPREDICTABLE** causes a trap exception that is taken to EL2.

In body text, the term **CONSTRAINED UNPREDICTABLE** is shown in SMALL CAPITALS.

See also [UNPREDICTABLE](#).

Containable error See [Contained error](#).

Contained error An error that is not uncontained or uncontainable.

Context switch The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

Context Synchronization event One of:

- Performing an ISB operation. An ISB operation is performed when an **ISB** instruction is executed and does not fail its condition code check.
- Exception entry.
- Return from an exception.
 - Exit from Debug state.
- Executing a **DCPS** instruction.
- Executing a **DRPS** instruction.

The effects of a Context synchronization event are:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- If halting is allowed, all Halting debug events that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions appearing in program order after an instruction that causes a Context synchronization event will have performed any part of their functionality until the Context synchronization event has occurred.
- All direct and indirect writes to System registers that are made before the Context synchronization event affect any instruction, including a direct read, that appears in program order after the instruction causing the Context synchronization event.
- All completed changes to the translation tables for entries that, before the change, were not permitted to be cached in a TLB, affect all instruction fetches that appear in program order after the instruction causing the Context synchronization event.
- The following invalidations, if completed before the Context synchronization event will affect all instructions that appear in program order after the Context synchronization event:
 - Instruction caches.
 - Branch predictors.
- All Non-cacheable writes that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.
- Changes to the Debug external authentication interfaces that are made before the Context synchronization event affect any instruction that appears in program order after the instruction causing the Context synchronization event.
- Ensuring that the **TSB** instruction is executed in the necessary order with respect to other instructions.
- See also:
 - Synchronization and debug exceptions.
 - [Synchronization and Halting debug events](#).
 - [Synchronization of memory-mapped registers](#).
 - **CONSTRAINED UNPREDICTABLE** behavior due to inadequate context synchronization.

Note

- The architecture requires that instructions that generate Context synchronization events do not appear to be executed speculatively, except that the performance monitor counters are permitted to reveal such speculation.
 - *Context synchronization events* were previously described as *context synchronization operations*.
-

Conventional memory Memory locations from which generic OSs and application runtimes expect to create allocations for general software use.

Core See [Processing element \(PE\)](#).

CoreSight Address Translation Unit A form of System MMU for trace streams.

Corrected error An error that is detected by hardware and that hardware has corrected.

Counter resolution See [Counter resolution](#).

CPBM Cache-Portion Bit Map.

CSU Cache-Storage Usage.

Data independent timing (DIT) The time that it takes to execute a piece of code where the time is not a function of the data being operated on. For more information, see [About PSTATE.DIT](#) and [About the DIT bit](#).

Debugger In most of this manual, *debugger* refers to any agent that is performing debug. However, some chapters or parts of this manual require a more rigorous definition, and define debugger locally. See:

- [Definition of a debugger in the context of self-hosted debug.](#)
- [Definition of a debugger in the context of self-hosted debug.](#)
- [Definition and constraints of a debugger in the context of external debug.](#)

Deferred error An error that has not been silently propagated but does not require immediate action at the producer. The error might have passed from the producer to a consumer.

Deprecated Something that is present in the Arm architecture for backward compatibility.

Features that are deprecated but are not **OPTIONAL** are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and **OPTIONAL**, in future versions of the Arm architecture.

Arm strongly recommends software to avoid using deprecated features. Where a feature is deprecated for performance reasons, Arm strongly recommends software to avoid using that feature because of the likelihood of the significant negative impact on performance.

See also [OPTIONAL](#).

Destructive instruction encoding A destructive instruction encoding is an instruction encoding where one of the source registers is also used as the destination register.

Detected error An error that has been detected and signaled to a consumer.

Detected Uncorrected error A detected error that has not been corrected and causes failure.

Digital signal processing (DSP) Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

Direct branch Branch instructions whose target is relative to the PC of the branch and the offset is determined solely from the opcode of the executed instruction.

See also [Indirect branch](#).

Direct Memory Access (DMA) An operation that accesses main memory directly, without the PE performing any accesses to the data concerned.

Direct read A direct read of a System register is a read performed by a System register access instruction in which the contents of the System Register addressed by {op1, Crm, Crn, op2}

For more information, see [Direct read](#).

See also [Direct write](#), [Indirect read](#), [Indirect write](#).

Direct write A direct write of a System register is a write performed by a System register access instruction.

For more information, see [Direct write](#).

See also [Direct read](#), [Indirect read](#), [Indirect write](#).

DMA See [Direct Memory Access \(DMA\)](#).

DNM See [Do-Not-Modify \(DNM\)](#).

Domain In the Arm architecture, *domain* is used in the following contexts.

Shareability domain Defines a set of observers for which the Shareability attributes make the data or unified caches transparent for data accesses.

Power domain Defines a block of logic with a single, common, power supply.

Memory regions domain

When using the Short-descriptor translation table format, defines a collection of Sections, Large pages and Small pages of memory, that can have their access permissions switched rapidly by writing to the *Domain Access Control Register* (DACR). Arm deprecates any use of memory regions domains.

Do-Not-Modify (DNM) Means the value must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same PE.

Double-precision value Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

Doubleword A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Doubleword-aligned Means that the address is divisible by 8.

Downstream The direction from Requesters to Completers.

DSB Data Synchronization Barrier.

DSP See [Digital signal processing \(DSP\)](#).

Effective value A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an *Effective value*:

- In some cases, the description of a control *a* specifies that when control *a* is active it causes a register control field *b* to be treated as having a fixed value for all purposes other than direct reads, or direct reads and direct writes, of the register containing control field *b*. When control *a* is active that fixed value is described as the *Effective value* of register control field *b*. For example, when the value of [HCR.DC](#) is 1, the *Effective value* of [HCR.VM](#) is 1, regardless of its actual value.

In other cases, in some contexts a register control field *b* is not implemented or is not accessible, but behavior of the PE is as if control field *b* was implemented and accessible, and had a particular value. In this case, that value is the *Effective value* of register control field *b*.

Note

Where a register control field is introduced in a particular version of the architecture, and is not implemented in an earlier version of the architecture, typically it will have an *Effective value* in that earlier version of the architecture.

- Otherwise, the *Effective value* of a register control field is the value of that field.

Endianness An aspect of the system memory mapping.

See also [Big-endian memory](#) and [Little-endian memory](#).

Error Correction Code (ECC) A code capable of detecting and correcting a number of errors.

Exception Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

Exception vector A fixed address that contains the address of the first instruction of the corresponding exception handler.

Executing processing element (PEe) The name given to an example PE that executes a specified instruction.

See also [Processing element \(PE\)](#).

Execution stream The stream of instructions that would have been executed by sequential execution of the program.

External abort An abort that is generated by the external memory system.

Fault An event that causes the program to deviate from the intended behavior, for example an invalid translation.

Field Replaceable Unit (FRU) A component or unit in a system that can be replaced without return to base.

First active element The First active element of an SVE vector or predicate register is the lowest numbered element that is an Active element.

Flat address mapping Is where the physical address for every access is equal to its virtual address.

Flush-to-zero mode A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and Intermediate results with zeros, without significantly affecting the accuracy of their final results.

FRU See [Field Replaceable Unit \(FRU\)](#).

General-purpose registers The registers that the base instructions use for processing:

- In AArch64 state the general-purpose registers are:
 - W0-W30 when accessed as 32-bit registers.
 - X0-X30 when accessed as 64-bit registers.
- In AArch32 state the general-purpose registers are R0-R14, that can also be described as R0-R12, SP, LR.

Note

Older documentation defines the AArch32 general-purpose registers as R0-R12, and the Arm core registers as R0-R12, SP, LR, and PC.

See also [High registers](#), [Low registers](#).

Generic Interrupt Controller (GIC) Arm system architecture interrupt controller for IRQ and FIQ interrupt exceptions.

Governing predicate The predicate register that is used to determine the Active elements of a predicated instruction is known as the Governing predicate for that instruction.

GIC See [Generic Interrupt Controller \(GIC\)](#).

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned Means that the address is divisible by 2.

High registers In AArch32 state, the general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

Note

In some contexts, *high registers* refers to R8-R15, meaning R8-R14 and the PC.

See also [General-purpose registers](#), [Low registers](#).

High vectors	An alternative location for the exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.
ICN	InterConnect Network.
IGNORED	Indicates that the architecture guarantees that the bit or field is not interpreted or modified by hardware. In body text, the term IGNORED is shown in SMALL CAPITALS.
Immediate and offset fields	Are unsigned unless otherwise stated.
Immediate value	A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many instructions can be used with an immediate argument.
IMP	An abbreviation used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.
IMPLEMENTATION DEFINED	Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations. In body text, the term IMPLEMENTATION DEFINED is shown in SMALL CAPITALS.
IMPLEMENTATION SPECIFIC	Means that the behavior is not architecturally defined, and might not be documented by an individual implementation. Used when there are several implementation options available and the option chosen does not affect software compatibility. In body text, the term IMPLEMENTATION SPECIFIC is shown in SMALL CAPITALS.
Imprecise exception	An exception that is not precise.
Index register	A register specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.
Indirect branch	Branch instructions that are not direct branches. See also Direct branch .
Indirect read	An Indirect System Register Read is a System Register Read which is not a Direct System Register Read Effect. For example, when an instruction uses a System register value to establish operating conditions, that use of the System register is an indirect read of the System register. For more information, including additional examples of indirect reads, see Indirect read . See also Direct read , Direct write , Indirect write .
Indirect write	An indirect write of a System register occurs when the contents of a register are updated by some mechanism other than a Direct write to that register. For example, an indirect write to a register might occur as a side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation performed by an external agent. For more information, see Indirect write . See also Direct read , Direct write , Indirect read .
Inline literals	These are constant addresses and other data items held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.
Intermediate physical address (IPA)	An implementation of virtualization, the address to which a Guest OS maps a VA. A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA. See also Physical address (PA) , Virtual address (VA) .
Interrupt	In a PE context, an asynchronous exception. There are three interrupt exceptions: IRQ, FIQ, and SError. IRQ and FIQ are always precise. In a system architecture context, an asynchronous event sent to a PE or GIC for processing as an interrupt exception.

Interworking	A method of working that permits branches between software using the T32 and A32 instruction sets.
IPA	See Intermediate physical address (IPA) .
kvm	Kernel-based Virtual Machine, an open-source software package that implements a type-2 hypervisor within Linux.
Latent error	An error that is present in a system but not yet detected.
Last active element	The Last active element of an SVE vector or predicate register is the highest numbered element that is an Active element.
Level	See Cache level .
Level of Coherence (LoC)	The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency. For more information, see Terms used in describing the cache maintenance instructions. See also Cache level , Point of coherency (PoC) .
Level of Unification, Inner Shareable (LoUIS)	The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable Shareability domain. For more information, see Terms used in describing the cache maintenance instructions. See also Cache level , Point of unification (PoU) .
Level of Unification, uniprocessor (LoUU)	For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE. For more information, see Terms used in describing the cache maintenance instructions. See also Cache level , Point of unification (PoU) .
Line	See Cache line .
Little-endian memory	Means that, for example: • A byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address. • A byte at a halfword-aligned address is the least significant byte in the halfword at that address. See also Big-endian memory , Endianness .
Load/store architecture	An architecture where data-processing operations only operate on register contents, not directly on memory contents.
LoC	See Level of Coherence (LoC) .
LoUIS	See Level of Unification, Inner Shareable (LoUIS) .
LoUU	See Level of Unification, uniprocessor (LoUU) .
Lockdown	See Cache lockdown .
Low registers	In AArch32 state, general-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers. See also General-purpose registers , High registers .
MBWU	Memory BandWidth Usage.
Memory barrier	See Memory barriers.
Memory coherency	The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

Memory element	An item of data in memory that is transferred to or from a vector element by an SVE load or store instruction. Each memory element has an access size and a type. The memory element access size is specified by each load and store instruction independently of the vector element size.
Memory Management Unit (MMU)	Provides detailed control of the part of a memory system that provides a single stage of address translation. Most of the control is provided using translation tables that are held in memory, and define the attributes of different regions of the physical memory map.
Memory Protection Unit (MPU)	A hardware unit whose registers provide simple control of a limited number of protection regions in memory.
Merging predication	When a predicated SVE instruction specifies merging predication, the Inactive elements of the destination register remain unchanged.
Memory system component	MSC. A function, unit, or design block in a memory system that can have partitionable resources. MSCs consist of all units that handle load or store requests issued by any MPAM Requester. These can include cache memories, interconnects, memory management units, memory channel controllers, queues, buffers, and rate adapters. An MSC can contain one or more resources that each can have zero or more resource partitioning controls. For example, a PE can contain several caches, each of which might have zero or more resource partitioning controls.
Memory system resource	A resource that affects the performance of software's use of the memory system and is either local to an MSC (such as cache-memory capacity) or non-local (such as memory bandwidth, which is present over an entire path, from Requester to Completer, that can pass through multiple MSCs).
MMR	Memory-mapped Register.
Microarchitecturally-finished	An operation that has finished all of its operational pseudocode, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers.
Microarchitecturally-unfinished	An operation that has not completed all of its operational pseudocode.
Miss	See Cache miss .
MMU	See Memory Management Unit (MMU) .
MPU	See Memory Protection Unit (MPU) .
Multi-copy atomicity	The form of atomicity described in Requirements for multi-copy atomicity. See also Atomicity , Single-copy atomicity .
NaN	Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a <i>quiet</i> NaN, that propagate through most floating-point operations, or a <i>signaling</i> NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the <i>IEEE Standard for Floating-point Arithmetic</i> . See also Quiet NaN , Signaling NaN .
Natural eviction	A natural eviction is an eviction that occurs in the course of the normal operation of the memory system, rather than because of an operation that explicitly causes an eviction from the cache, such as the execution of a cache maintenance instruction. Typically, a natural eviction occurs when the caching algorithm requires data to be cached but the cache does not have room for that data.
Non-fault load	SVE provides a Non-fault option for some SVE vector load instructions. This option causes all memory access faults to be suppressed. Instead, the FFR is updated to indicate which of the Active elements were not successfully loaded.
Observer	An observer is one of: • The mechanism inside a PE that generates Implicit Translation Table Descriptor (TTD) Memory Effects. • The mechanism inside a PE that generates Implicit Instruction Memory Read Effects. • The mechanism inside a PE that generates Explicit Memory Effects, Tag Memory Effects, Barrier Effects, Instruction Fetch Barrier Effects, TLBI Effects, DC Effects, IC Effects, Fault Effects, or GCS Effects.

- The mechanism inside a PE that generates Memory Effects for SPE.
- The mechanism inside a PE that generates Memory Effects for TRBE.
- The mechanism inside a PE that generates Memory Effects for HDBSS.
- The mechanism inside a PE that generates Memory Effects for HACDBS.
- Any mechanism outside the PE that can generate Memory Effects, for example, a peripheral device, an SMMU, or a GIC.

Obsolete Obsolete indicates something that is no longer supported by Arm. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

Offset addressing Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

OPTIONAL When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a feature to be included in a new implementation only if there is a known backward-compatibility reason for the inclusion of the feature. A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.
- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALL CAPITALS.

Note

Do not confuse these Arm-specific uses of OPTIONAL with other uses of *optional*, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
 - Behavior determined by an implementation choice, for example the optional byte order reversal in an Armv7-R implementation, where the [SCTLR.IE](#) bit indicates the implemented option.
-

See also [Deprecated](#).

Other-multi-copy atomic This is a Memory Write effect from an observer that, if observed by a different observer, is then observed by all other observers that access the location coherently.

PA See [Physical address \(PA\)](#).

PCIe See [Peripheral Component Interconnect Express \(PCI Express or PCIe\)](#).

PE See [Processing element \(PE\)](#).

Peripheral Component Interconnect Express (PCI Express or PCIe) A high-speed serial computer expansion bus standard maintained and developed by the PCI Special Interest Group.

PEe The executing processing element. See [Executing processing element](#).

PFA See Predictive Failure Analysis (PFA).

Physical address (PA) An address that identifies a location in the physical memory map.

See also [Intermediate physical address \(IPA\)](#), [Virtual address \(VA\)](#).

Point of coherency (PoC) See Terms used in describing the cache maintenance instructions for the AArch64 state and [Terminology for Clean, Invalidate, and Clean and Invalidate instructions](#) for the AArch32 state.

Point of Deep Persistence (PoDP) See Terms used in describing the cache maintenance instructions.

Point of encryption (PoE) See Terms used in describing the cache maintenance instructions.

Point of persistence (PoP) See Terms used in describing the cache maintenance instructions.

Point of physical aliasing (PoPA) See Terms used in describing the cache maintenance instructions.

Point of physical storage (PoPS) See Terms used in describing the cache maintenance instructions.

Point of unification (PoU) See Terms used in describing the cache maintenance instructions for the AArch64 state and [Terminology for Clean, Invalidate, and Clean and Invalidate instructions](#) for the AArch32 state.

Poison atomicity The form of atomicity described in Clearing Deferred errors from poisoned Locations.

See also [Atomicity](#).

Poison granule A region of naturally-aligned memory affected by poison.

See Error propagation for information on poison.

Post-indexed addressing Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

Precise exception An exception where the exception handler receives the state of the PE and the state of the memory system consistent with the PE having executed all of the instructions up to, but not including, the point in the instruction stream where the exception was taken. The state of the PE and the state of the memory do not include instructions that occurred after this point.

Prefetching Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

Note

The Prefetch Abort exception can be generated on any instruction fetch, and is not limited to speculative instruction fetches.

See also [Simple sequential execution](#).

Prefixed instruction The instruction that immediately follows a MOVPRFX instruction in program order.

Pre-indexed addressing Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

Processing element (PE) The abstract machine defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Protection region A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit See [Memory Protection Unit \(MPU\)](#).

Pseudo-instruction UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>, <Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>, <Rm>, #<n>`.

PSTATE An abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

See also [Condition flags](#).

Quadword A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

Quadword-aligned Means that the address is divisible by 16.

Quiet NaN	A NaN that propagates unchanged through most floating-point operations. See also NaN, Signaling NaN.
RAO	See Read-As-One (RAO) .
RAZ	See Read-As-Zero (RAZ) .
RAO/SBOP	In versions of the Arm architecture before Armv8, Read-As-One, Should-Be-One-or-Preserved on writes. From the introduction of the Armv8 architecture, RES1 replaces this description. See also UNK/SBOP, Read-As-One (RAO), RES1, Should-Be-One-or-Preserved (SBOP).
RAO/WI	Read-As-One, Writes Ignored. Hardware must implement the field as Read-As-One, and must ignore writes to the field. Software can rely on the field reading as all 1s, and on writes being ignored. This description can apply to a single bit that reads as 1, or to a field that reads as all 1s. See also Read-As-One (RAO).
RAZ/SBZP	In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes. From the introduction of the Armv8 architecture, RES0 replaces this description. See also UNK/SBZP, Read-As-Zero (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).
RAZ/WI	Read-As-Zero, Writes Ignored. Hardware must implement the field as Read-As-Zero, and must ignore writes to the field. Software can rely on the field reading as all 0s, and on writes being ignored. This description can apply to a single bit that reads as 0, or to a field that reads as all 0s. See also Read-As-Zero (RAZ).
Read-allocate cache	A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.
Read-As-One (RAO)	Hardware must implement the field as reading as all 1s. Software: • Can rely on the field reading as all 1s. • Must use a SBOP policy to write to the field. This description can apply to a single bit that reads as 1, or to a field that reads as all 1s. See also RAO/SBOP, RAO/WI, RES1.
Read-As-Zero (RAZ)	Hardware must implement the field as reading as all 0s. Software: • Can rely on the field reading as all 0s • Must use a SBZP policy to write to the field. This description can apply to a single bit that reads as 0, or to a field that reads as all 0s. See also RAZ/SBZP, RAZ/WI, RES0.
Read-to-Clear (RC)	A read of the field clears the field to 0 once the read is complete.
Read, modify, write	In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields updated in that register, and the new value written back.
Recoverable error	A contained error that must be corrected to allow the correct operation of the system or smaller parts of the system to continue.

Reliability	Continuity of correct service.
Requester	An agent in a computing system that is capable of initiating memory transactions. See also Completer .
RES0	A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors. Within the architecture, there are some cases where a register bit or field: • Is RES0 in some defined architectural context. • Has different defined behavior in a different architectural context. Note • RES0 is not used in descriptions of instruction encodings. • Where an AArch32 System register is Architecturally mapped to an AArch64 System register, and a bit or field in that register is RES0 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES0 for fields in read/write registers is:

If a bit is RES0 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
 - Reads of the bit always return 0.
 - Writes to the bit are ignored.
2. The bit can be written. In this case:
 - An indirect write to the register sets the bit to 0.
 - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
 - A direct write to the bit must update a storage location associated with the bit.
 - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a bit is RES0 only in some contexts

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.

- While the use of the register is such that the bit is described as **RES0**, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as **RES0**, another context in which the same bit is defined as **RES1**, and no context in which the bit is defined as a functional bit, then it is **IMPLEMENTATION DEFINED** whether:

- Writes to the bit are ignored, and reads of the bit return an **UNKNOWN** value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The **RES0** description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, **RES0** indicates that the bit reads as 0, but software must treat the bit as **UNKNOWN**.
- For a write-only bit, **RES0** indicates that software must treat the bit as **SBZ**.

A bit that is **RES0** in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an **SBZP** policy to write to the bit.

The definition of a **RES0** field in an architecturally-defined data structure that is held in memory is:

- A read of a **RES0** bit returns the value last written.
- The value of a **RES0** bit has no effect on the operation of the PE or any other hardware implementing the architecture such as the GIC or the SMMU, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.
- For an implicit memory write effect to the field by a PE or for a write to the field by any other hardware implementing the architecture such as the GIC or the SMMU, each bit defined as **RES0** is set to 0.
- For a write to the data structure by a PE, or any other hardware implementing the architecture such as the GIC or the SMMU, if the write is specified to only update specific fields, the hardware preserves the value of other **RES0** bits in the same data structure. This applies, for example, to Hardware updates to the translation tables.

This **RES0** description can apply to a single bit, or to a field for which each bit of the field must be treated as **RES0**.

In body text, the term **RES0** is shown in **SMALL CAPITALS**.

See also [Read-As-Zero \(RAZ\)](#), [RES0H](#), [RES1](#), [Should-Be-Zero-or-Preserved \(SBZP\)](#), [UNKNOWN](#).

RES0H

A reserved bit or field with **SBZP**. This behavior uses the hardwired to 0 subset of the **RES0** definition.

See also [Read-As-Zero \(RAZ\)](#), [RES0](#), [RES1](#), [Should-Be-Zero-or-Preserved \(SBZP\)](#).

RES1

A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is **RES1** in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- **RES1** is not used in descriptions of instruction encodings.
 - Where an AArch32 System register is [Architecturally mapped](#) to an AArch64 System register, and a bit or field in that register is **RES1** in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.
-

This means the definition of RES1 for fields in read/write registers is:

If a bit is RES1 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
 - Reads of the bit always return 1.
 - Writes to the bit are ignored.
2. The bit can be written. In this case:
 - An indirect write to the register sets the bit to 1.
 - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
 - A direct write to the bit must update a storage location associated with the bit.
 - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a bit is RES1 only in some contexts

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

Note

As indicated in this list, this value might be written by an indirect write to the register.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

The definition of a RES1 field in an architecturally-defined data structure that is held in memory is:

- A read of a RES1 bit returns the value last written.
- The value of a RES1 bit has no effect on the operation of the PE or any other hardware implementing the architecture such as the GIC or the SMMU, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.
- For an implicit memory write effect to the field by a PE or for a write to the field by any other hardware implementing the architecture such as the GIC or the SMMU, each bit defined as RES1 is set to 1.
- For a write to the data structure by a PE, or any other hardware implementing the architecture such as the GIC or the SMMU, if the write is specified to only update specific fields, the hardware preserves the value of other RES1 bits in the same data structure. This applies, for example, to Hardware updates to the translation tables.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also [Read-As-One \(RAO\)](#), [RES0](#), [RES0H](#), [Should-Be-One-or-Preserved \(SBOP\)](#), [UNKNOWN](#).

Reserved

Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions described as reserved are:
 - In an RW or WO register, RES0.
 - In an RO register, UNK.

See also [CONSTRAINED UNPREDICTABLE](#), [RES0](#), [RES0H](#), [RES1](#), [UNDEFINED](#), [UNK](#), [UNPREDICTABLE](#).

Resource partition The collection of MPAM resource control settings associated with a software environment and identified by the combination of a physical PARTID space and a partition number.

RESS Reserved, Sign extended. A register value is extended by copying the sign bit into all of the reserved bits to the left of the most significant bit of the field. The values of these bits are identical to the most significant bit of the value being extended.

Within the architecture, a register bit or field can be treated:

- As RESS in few defined architectural contexts.
- In a different defined behavior in other architectural contexts.

This means the definition of RESS for fields in read/write registers is:

If a field [Q:P] is RESS in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired as a sign-extension of bit [P-1]. In this case:
 - Reads of the bit always return the value of bit [P-1].
 - Writes to the bit are ignored.
2. The bit can be written. In this case:

- An indirect write to the register sets the bit to the same value as bit [P-1].
- A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.
If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RESS bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a field [Q:P] is RESS only in some contexts

For a bit in a read/write register, when the bit is described as RESS:

- An indirect write to the register sets the bit to the same value as bit [P-1].
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

Note

As indicated in this list, this value might be written by an indirect write to the register.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RESS, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RESS description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RESS indicates that the bit reads as the value of bit [P-1], but software must treat the bit as UNKNOWN.
- For a write-only bit, RESS indicates that software must treat the bit as **SB0** or **SBZ** as necessary.

A bit that is RESS in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as [P-1].
- Must use a **SBOP** or **SBZP** policy to write to the bit.

This RESS description can apply to a single bit, or to a field for which each bit of the field must be treated as RESS.

- Rewind point** A rewind point is a point in the program flow to which execution can return if all subsequent execution is found to have been incorrectly speculatively executed.
- RISC** Reduced Instruction Set Computer.
- Rounding error** The value of the rounded result of an arithmetic operation minus the exact result of the operation.
- Rounding mode** Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic*, see [Floating-point standards, and terminology](#).
- Saturated arithmetic** Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from $+2^{31}-1$ to -2^{31} or vice versa.
- SBO** See [Should-Be-One \(SBO\)](#).
- SBOP** See [Should-Be-One-or-Preserved \(SBOP\)](#).
- SBZ** See [Should-Be-Zero \(SBZ\)](#).
- SBZP** See [Should-Be-Zero-or-Preserved \(SBZP\)](#).
- Scalar base register** A scalar base register refers to an AArch64 general-purpose register, X0-X30, or the current stack pointer, SP.
- Scalar index register** A scalar index register refers to an AArch64 general-purpose register, X0-X30, or for certain instructions, XZR.
- Security hole** A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.
- See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).
- Self-modifying code** Code that writes one or more instructions to memory and then executes them. When using self-modifying code, you must use cache maintenance and barrier instructions to ensure synchronization. For more information, see [Caches and memory hierarchy](#).
- SError exception** An asynchronous exception in the Arm architecture.
- Serviceability** The ability to undergo modifications and repairs.
- Service failure mode** A mode entered to reduce the severity of an error.
- Set** See [Cache sets](#).
- Should-Be-One (SBO)** Hardware must ignore writes to the field.
- Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.
- This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.
- See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).
- Should-Be-One-or-Preserved (SBOP)** From the introduction of the Armv8 architecture, the description [Should-Be-One-or-Preserved \(SBOP\)](#) is superseded by [RES1](#).
- Hardware must ignore writes to the field.
- When writing this field, software must either write all 1s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).

Should-Be-Zero (SBZ) Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).

Should-Be-Zero-or-Preserved (SBZP) From the introduction of the Armv8 architecture, the description [Should-Be-Zero-or-Preserved \(SBZP\)](#) is superseded by [RES0](#).

Hardware must ignore writes to the field.

When writing this field, software must either write all 0s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).

Signaling NaN An Invalid Operation floating-point exception occurs whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

See also [NaN](#), [Quiet NaN](#).

Signed immediate and offset fields Are encoded in two's complement notation unless otherwise stated.

Silent Data Corruption (SDC) An error that is not detected by hardware or software.

SIMD Single-Instruction, Multiple-Data.

The SIMD instructions in AArch32 state are:

- The instructions summarized in [Parallel addition and subtraction instructions](#).
- The Advanced SIMD instructions summarized in [Advanced SIMD and floating-point instructions](#), when operating on vectors.

Simple sequential execution The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Single-copy atomicity The form of atomicity described in Properties of single-copy atomic accesses.

See also [Atomicity](#), [Multi-copy atomicity](#).

Single-precision value A 32-bit word that is interpreted as a basic single-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

SMMU System Memory Management Unit.

Spatial locality The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

Special-purpose register One of a specified set of registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

- Special purpose registers specifies the AArch64 Special-purpose registers.
- [AArch32 general-purpose registers, the PC, and the Special-purpose registers](#) lists the AArch32 Special-purpose registers.

Speculative A speculative operation is any of the following:

- An operation performed by the PE before the PE has determined that the operation is required to be performed in a simple sequential execution.
- An operation performed by the PE that is not required to be performed in a simple sequential execution.
- An operation performed by the PE before the PE has architecturally resolved the architectural state of inputs or outputs of the operation.

Examples of speculative operations are:

- Operations that are generated by instructions that appear in the Execution stream after a branch that is not architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after an instruction where a synchronous exception condition has not been architecturally resolved.
- Operations that are generated by conditional instructions for which the conditions for the instruction have not been architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after the point at which a precise asynchronous exception will be taken.
- Operations generated by instructions for which one or more of the arguments come from a register that has not been architecturally resolved.
- Operations generated by the hardware that are not directly generated by any instructions appearing in the Execution stream.
- A memory effect (M2) that appears in program order after a memory effect (M1) where all of the following apply:
 - M1 is locally-ordered-before M2.
 - M1 has not been architecturally resolved.
- Read accesses generated for a translation table walk for which the granule protection check for the address being accessed has not been architecturally resolved.

See also [Execution stream](#).

SPI Shared Peripheral Interrupt.

Superpriority An attribute that is used to denote virtual and physical IRQ and FIQ interrupts as non-maskable. IRQ and FIQ interrupts with Superpriority can be taken under certain conditions where usually they would be masked by the PSTATE.{I, F} bits.

System Control Processor A PE dedicated to system control and monitoring.

T32 instruction One or two halfwords that specify an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in T32 state. T32 instructions must be halfword-aligned.

T32 instructions were previously called Thumb instructions.

See also [A32 instruction](#), [A64 instruction](#), [T32 state](#).

T32 state The AArch32 Instruction set state in which the PE executes T32 instructions.

T32 state was previously called Thumb state.

See also [A32 state](#), [T32 instruction](#).

Temporal locality The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

Thumb instruction See [T32 instruction](#).

Unaligned An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Unallocated Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as **CONSTRAINED UNPREDICTABLE**, **UNDEFINED**, **UNPREDICTABLE**, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

See also [CONSTRAINED UNPREDICTABLE](#), [UNDEFINED](#), [UNPREDICTABLE](#).

Uncontainable error See [Uncontained error](#).

Uncontained error An error that has been, or might have been, silently propagated.

Uncorrected error An error caused by a fault which was not removed.

UNDEFINED Indicates cases where an attempt to execute a particular encoding bit pattern generates an exception, that is taken to the current Exception level, or to the default Exception level for taking exceptions if the **UNDEFINED** encoding was executed at EL0. This applies to:

- Any encoding that is not allocated to any instruction.
- Any encoding that is defined as never accessible at the current Exception level.
- Some cases where an enable, disable, or trap control means an encoding is not accessible at the current Exception level.

If the generated exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

Note

On reset, the default Exception level for taking exceptions from EL0 is EL1. However, an implementation might include controls that can change this, effectively making EL1 inactive. See the description of the Exception model for more information

In body text, the term **UNDEFINED** is shown in SMALL CAPITALS.

See also [Undefined Instruction exception](#).

Undetected error See [Latent error](#).

Unified cache Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing Means addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

UNK An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

See also [UNKNOWN](#).

UNK/SBOP Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also [Read-As-One \(RAO\)](#), [Should-Be-One-or-Preserved \(SBOP\)](#), [UNKNOWN](#).

UNK/SBZP Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also [Read-As-Zero \(RAZ\)](#), [Should-Be-Zero-or-Preserved \(SBZP\)](#), [UNKNOWN](#).

UNKNOWN An UNKNOWN value does not contain valid data, and can vary from implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE, are not CONSTRAINED UNPREDICTABLE, and do not return UNKNOWN values.

An UNKNOWN value can vary from moment to moment, and instruction to instruction, unless it has previously been assigned, other than at reset, to one of the following registers:

- Any of the general-purpose registers.
- Any of the Advanced SIMD and floating-point registers.
- Any of the Scalable Vector Extension registers.
- Any of the PSTATE N, Z, C, or V flags.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALL CAPITALS.

See also [CONSTRAINED UNPREDICTABLE](#), [UNDEFINED](#), [UNK](#), [UNPREDICTABLE](#).

UNPREDICTABLE Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

Execution at Non-secure EL1 or EL0 of an instruction that is UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term UNPREDICTABLE is shown in SMALL CAPITALS.

See also [CONSTRAINED UNPREDICTABLE](#), [UNDEFINED](#).

Upstream The direction from Completers to Requesters.

VA See [Virtual address \(VA\)](#).

VFP In Armv7, an extension to the Arm architecture, that provides single-precision and double-precision floating-point arithmetic.

Vector A one-dimensional array of vector elements of the same size and data type.

Vector element Individual subdivisions of a vector register that can be 8, 16, 32, 64 or 128 bits in size. The vector element size and data type is specified independently by each instruction.

Virtual address (VA) An address generated by an Arm PE. This means it is an address that might be held in the program counter of the PE. For a PMSA implementation, the virtual address is identical to the physical address.

See also [Intermediate physical address \(IPA\)](#), [Physical address \(PA\)](#).

VM Virtual Machine.

VMM Virtual Machine Monitor. An alias for “hypervisor”.

Virtual PARTID A PARTID in MPAM that can be used by a virtual machine (VM). Virtual PARTIDs are mapped into physical PARTIDs using the virtual partition mapping entries in the MPAMVPM0 - MPAMVPM7 registers.

Watchpoint A debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

Way	See Cache way .
WI	Writes Ignored. In a register that software can write to, a WI attribute applied to a bit or field indicates that the bit or field ignores the value written by software and retains the value it had before that write. See also RAO/WI, RAZ/WI, RES0, RES1.
Word	A 32-bit data item. Words are normally word-aligned in Arm systems.
Word-aligned	Means that the address is divisible by 4.
Write-allocate cache	A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.
Write-back cache	A cache in which when a cache hit occurs on a write access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in the next level of cache or memory. Any such data is written back to the next level of cache or memory when the cache line is cleaned. Another common term for a write-back cache is a <i>copy-back cache</i>.
Write-One-to-Clear (W1C)	Writes of 0 to the bit are ignored. A write of 1 clears the bit to 0.
Write-One-to-Set (W1S)	Writes of 0 to the bit are ignored. A write of 1 sets the bit to 1.
Write-Through cache	A cache in which when a cache hit occurs on a write access, the data is written both to the cache and to the next level of cache or memory.
Write buffer	A block of high-speed memory that optimizes stores to main memory.